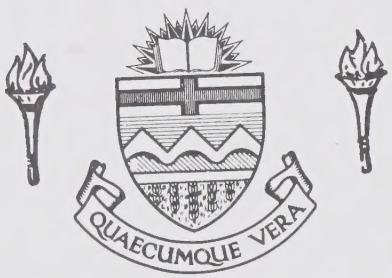


THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR Sundaravarathan R. Iyengar

TITLE OF THESIS Interconnecting multiple processors in a shared I/O fashion

DEGREE FOR WHICH THESIS WAS PRESENTED Master of Science

YEAR THIS DEGREE GRANTED September, 1982

THE UNIVERSITY OF ALBERTA

Interconnecting multiple processors in a shared I/O fashion

by

Sundaravarathan R. Iyengar

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF Master of Science

Department of Computing Science

EDMONTON, ALBERTA

Fall, 1982

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled Interconnecting multiple processors in a shared I/O fashion submitted by Sundaravarathan R. Iyengar in partial fulfilment of the requirements for the degree of Master of Science.

Dedicated to my parents

ABSTRACT

A special purpose multiple processor system, the Shared I/O, is discussed. It is a method of organizing multiple processors such that all processors perform identical operations. Due to the nature of the structure, it is important that the communication between the processors be managed in as short time span as possible. Two designs that supervise the inter-processor communications in such an environment are presented. The concept of shared I/O is also compared with other schemes used for similar purposes, in particular, pipelining.

ACKNOWLEDGEMENTS

I'd like to thank my supervisor, Dr. Rick Heuft. Without his valuable guidance and constant encouragement this thesis would not have been possible.

I thank Steve Sutphen for his help during the design phase of this thesis. I am thankful to Atul Gupta for many constructive discussions. My thanks are due to John de Haan and Farid Lahdiri for reading an earlier version of this thesis and offering helpful suggestions. Thanks are also due to Darrell Makarenko for his help in preparing the final version of this thesis.

Table of Contents

	Page
CHAPTER 1 Introduction	1
1.1 An Overview	1
1.2 Shared I/O and other multiple-processor systems	2
1.2.1 Shared I/O in a nutshell	3
1.2.2 The aspect of memory	4
1.2.3 The aspect of cooperation	7
1.3 Summary	10
CHAPTER 2 Shared I/O Organization	14
2.1 Introduction	14
2.2 Specifics of Shared I/O	15
2.3 Summary	21
CHAPTER 3 Comparison with pipeline structures ..	32
3.1 Performance Considerations	33
3.1.1 Throughput Considerations	33
3.1.2 Efficiency Considerations	34
3.2 Systolic Arrays: an example	37
3.3 Other Aspects	40
3.4 Summary	44
CHAPTER 4 Inter-Processor Communication Mechanisms	50
4.1 Selection of a Processor	52
4.2 Operation of the PIO	56

4.2.1 Terminology 57
4.2.2 Data transfer operations using PIO .. 59
4.2.2.1 Send: OUT (nn), A 59
4.2.2.2 Receive: IN A,(nn) 60
4.3 The HOLD controller 61
4.3.1 General operation 61
4.3.2 Problems and solutions 66
4.3.3 Technical details 69
4.3.4 Testing of the HOLD controller 71
4.4 The FIFO circuit 72
4.4.1 The Am2812 73
4.4.2 Operation of the FIFO circuit 75
4.5 A method of using internal buffer 77
4.6 Asynchronous processors in shared I/O .. 81
4.7 Summary 85
CHAPTER 5 Conclusions 106
5.1 Summary 106
5.2 Future work 108
References 110

List of Tables

Page

3.1 Comparison of shared I/O and pipeline 45

4.1 Comparison of DMA and FIFO techniques 87

List of Figures

	Page
1.1 Shared I/O and other multiple processor systems	12
1.2 n processors in the shared I/O configuration	13
2.1 Ith iteration of program (2.1)	23
2.2 Scheduling adjacent processors	24
2.3 Ith iteration of equation 2.4	25
2.4 Scheduling adjacent processors	26
2.5 Interleaved data transfers	27
2.6 The case when Trs < Ts	28
2.7 An instance of equation 2.4	29
2.8 Another instance of equation 2.4	30
2.9 Yet anoher instance of equation 2.4	31
3.1 4 processors in pipeline configuration	46
3.2 Systolic cell and shared I/O processor for FIR filter	47
3.3 FIR filter expressed as a maxtrix multiplication	48
3.4 Systolic cell arrangement for FIR filter	48
3.5 First seven steps of the FIR algorithm	49
4.1 Communication protocol between P(i) and P(i+1)	88
4.2 Z80-PIO as connected to P(i)	89
4.3 Input/Output cycles of Z80 CPU	90
4.4 Timing relations on OUT instruction for the PIO	91

4.5 Timing relations on IN instruction for the PIO .. 92

4.6 HOLD protocol 93

4.7a Timing relations on OUT instruction for the HOLD controller with no buffer full condition ... 94

4.7b Timing relations on IN instruction for the HOLD controller with no buffer empty condition .. 95

4.8a Timing relations on OUT instruction for the HOLD controller with buffer full condition 96

4.8b Timing relations on IN instruction for the HOLD controller with buffer empty condition 97

4.9 HOLD controller 98

4.10 Premature release of P(i) from wait state during receive 99

4.11 The problem of deadlock 100

4.12 Dynamic testing program for HOLD controller .. 101

4.13 Timing diagram for Am2812 102

4.14 Block diagram for FIFO circuit 103

4.15 Timing diagram for FIFO circuit 104

4.16 FIFO circuit 105

CHAPTER 1

Introduction

1.1. An Overview

In the recent past, many computer systems have been designed specifically to implement a single algorithm or a group of similar algorithms. These special purpose hardware units are used for algorithms that require an extensive amount of time to execute on a conventional computer. Examples include systems developed for computer assisted tomography [BRO076, SWAR82], analyzing data from satellites [ONOE81] and real time processing of signals and images [RABI75].

This thesis presents, a design of yet another special purpose system, referred to as the Shared I/O organization, developed by Heuft [HEUF80]. Shared I/O is a method of organizing multiple processors such that all processors perform identical operations. This characteristic makes the shared I/O well suited for implementation using commercially available LSI components such as microprocessors. Due to the nature of the structure (to be discussed later), it is of paramount importance that the communication between various modules within the system be managed in as short time span as possible. Two designs that supervise the inter-processor communications

in such an environment are described. The concept of Shared I/O is also compared with other schemes used for similar purposes, like pipelining.

The next section relates shared I/O with the other multiple-processors systems available today. It is followed by a discussion of the shared I/O. The subsequent chapter compares shared I/O with pipelining and systolic arrays. Systolic arrays are highly regular cellular structures that work on the principle of pipeline yet have a close resemblance to shared I/O. It is followed by a description of the two methods of managing inter-processor communications. Conclusions and recommendations complete the thesis.

1.2. Shared I/O and other multiple-processor systems

Since the time of Unger [UNGE58], there has been a significant increase in the number of computer systems which attempt to improve the performance to cost ratio. Most of them make use of multiple processors cooperating with each other to achieve the desired performance levels. Since the shared I/O scheme also uses multiple processors, there is a need to present a clear idea of what shared I/O is in relation to other systems.

There are two approaches: one is based on a particular architectural feature, the memory and the other, on

the degree of cooperation among the processors in the system. We discuss the above in the following sections. Figure 1.1 illustrates the discussion to follow in a graphical manner. For completeness, a brief description of each node in the graph is given which includes few representative examples. Before we go into any detail we first briefly describe the shared I/O organization.

1.2.1. Shared I/O in a nutshell

The shared I/O organization is a special purpose system designed to implement a certain class of algorithms. It consists of multiple processors that cooperate on a computation. Figure 1.2 shows a set of processors connected in the shared I/O configuration. Each processor has its own memory that is not accessible to any other processor in the system. A simple interconnection scheme connects adjacent processors together to form the communication link. The first and last processors are considered adjacent. In shared I/O, processor(i) sends values to processor(i+1) and receives from processor(i-1). There are only two I/O devices for the entire system and they are shared by all the processors (hence the name shared I/O). The major envisaged use of the shared I/O is its function as a peripheral to a main computer.

1.2.2. The aspect of memory

The processors in a multiple processor system can either share a common memory or operate from within their own private memories. The multiprocessor as defined by Enslow [ENSL77] is a shared memory system. It consists of two or more processors sharing a main memory and I/O devices under the supervision of a single operating system. It also has the capability to allow interaction among the processors both at software and hardware levels. C.mmp [WULF72] and Cm* [SWAN77] are two well known models of such a multiprocessor. The shared I/O, though it allows the sharing of I/O devices, does not permit the sharing of memory.

When the memories attached to the processors are separate from each other, we can have several distinct multiple processor systems. There are distributed systems and computer networks.

Enslow defines a distributed system to be characterized by five components [ENSL78]:

(1) it has a number of processors which can be dynamically assigned to specific tasks;

(2) the processors are physically distributed and interact with each other over a communication network;

(3) the distribution of the processors and the computation they perform is transparent to the user. That is, a user is unaware of the distribution of the system and thus is unable to specify the processor he would like to use for a particular computation;

(4) the processors cooperate with each other on a computation though they remain autonomous with respect to each other. That is, they retain the right to refuse a service request from other processors.

(5) and finally, each processor has its own local operating system; there is also a common operating system that supervises and integrates the operations of the entire system;

HXDP developed at the Honeywell is an example of an experimental distributed system [JENS78].

Shared I/O conforms to the above definition with respect to items 2 and 4. The processors in the shared I/O are autonomous, because they may not grant their neighbors permission to communicate with them until they are ready to do so. For instance, processor P(i) may not process a send from P(i-1) until is ready to receive the data. As for item 1, tasks are not dynamically assigned to the processors in the shared I/O during a computation; they are determined at the beginning of the execution. Similarly, item 3 is not applicable because, the processors in the

shared I/O perform identical tasks. Thus, the shared I/O is not a distributed system.

Computer networks are similar to distributed systems except that there is no system transparency. The user must be knowledgeable about the resources available elsewhere and specify explicitly how to access them. Also there is no high level operating system that controls and integrates the underlying hardware. To state it otherwise, computer networks are designed for resource sharing rather than sharing of the computational load. Shared I/O is not a computer network because its main objective in introducing cooperation among the processors is to share the computational load.

Other multiple processor systems we have not yet considered so far are systems that exhibit a master slave relationship. These are neither multiprocessors [ENSL74] nor distributed systems nor computer networks [ENSL78]. Shared I/O could be classified as a member of this category in that it operates as a slave to a master. Within the slave there could be multiple processors cooperating on a task. For example, the pipelined central processor in TI-ASC operates as a slave to the peripheral processor, which executes the operating system [WATS72]. The peripheral processor analyzes a job and initiates the central processor on the execution of the job after furnishing it with the required data. In a similar fashion,

pipelined array processors such as Floating Point Systems AP family, are designed to be peripheral devices that act on a command from the master. [BERN82] gives an example of how a matrix multiplication operation is carried out on the FPS AP. The master, referred to as the host, transfers all the data necessary for the multiplication to the array processor and receives the results from the latter. The shared I/O can be programmed to do operations similar to that of the FPS AP family. The difference between the two is the methods of computation employed. Shared I/O exploits parallelism in the given operation while FPS AP uses pipelining.

1.2.3. The aspect of cooperation

When there are multiple processors in a system they usually cooperate among each other to share the computational workload. There are three levels of cooperation a multiple processor system may exhibit: job, task or instruction. At the job level, different processors are initiated with separate jobs independent from each other. Traditional multiprogramming as done on a multiprocessor is an example. The experimental version of Michigan Terminal System (MTS), is a multiprogramming system built around two IBM 360/65MP processors [SROD78]. At any given time, the processors are executing different jobs originating from separate users. The cooperation between the

processors is in the form of observing the etiquettes of sharing common resources, such as memory or an I/O device. Thus communication needs are at a minimum. The PRIME computer [BASK72] is an example of a private memory multiple processor system in which the processors cooperate at the job level.

At the task level of cooperation, a job is partitioned into several tasks that are distributed among the processors. Since all the tasks must cooperate with each other, it is necessary for some communication mechanisms to be established between the different tasks so that the values produced by one may be transferred to another. The communication needs at this level of cooperation tend to be moderate because the job is partitioned in a way that would minimize the communication overhead. C.mmp is an example of such a multiprocessor, consisting up to sixteen processors with task level cooperation [WULF72]. Cm* is another instance of task level cooperation in a multiprocessor [SATY80]. All useful computation in Cm* is done through task forces, which are activities that may execute in parallel. HXDP [JENS78] is an example of private memory system that exhibits cooperation at the task level.

At the instruction level of cooperation, independent instructions are executed in parallel on different processors. There are three methods of organizing the instruction level cooperation: parallel, pipelined or overlapped

operation. Parallel execution refers to performing independent operations simultaneously on different processor. In a pipeline, the execution function is subdivided such that the intermediate results move from one stage to another for application of another subfunction. For example, the floating point addition on the TI-ASC [WATS72] is a pipelined function. Overlapped processing is different from pipelining in that the evaluation of the function may require a different sequence of subfunctions depending upon the dynamic state of the system [KOGG81]. The instruction overlapping in IBM 360 model 91 [ANDE67] is an excellent example. Pipelined vector processors such as the TI-ASC and CDC STAR utilize the parallelism among identical operations that have to be performed on large vectors of data [RAMA77]. STARAN and PEPE are examples of parallel execution of independent instructions in which the operands are broadcast to a set of similar processors [RUDO72, EVEN73]. MPP and ASPRO are improvements over STARAN developed at the Goodyear Aerospace that make use of advanced technology for increased speed of operation [BATC80, BATC82]. The concept of data flow is yet another example which employs cooperation at the instruction level. In a data flow computer, the execution of each instruction depends upon some other instruction which produces its inputs [TREA82]. Shared I/O is a multiple processor system that cooperates at the instruction level. In it, each processor may depend upon values produced by

its adjacent processor for its proper operation. High speed communication facilities are required at this level of cooperation since the intermediate results have to be passed among the processors on a time scale comparable to that of instruction execution time.

1.3. Summary

To summarize, we have presented the shared I/O organization in relation to the multiple processor computer systems available today. It is a special purpose system that cooperates at the instruction level. It has no common memory and it uses shared I/O devices. It can be used as a peripheral processor to a main computer. The next chapter presents the shared I/O in a greater detail. It develops specific relations that characterize the system.

Before we conclude this chapter, we would like to say a few words about special purpose systems in general. We have mentioned that shared I/O is a *special* purpose system. By being special, it circumvents most of the problems that are commonly related to general purpose systems. For example, the interconnection scheme used for shared I/O is very simple because its requirements are very specialized. Communication is in one direction only and between adjacent processors. In contrast, the interconnection network required in a general purpose multiple processor system

can be very complex and hence expensive [FENG81]. Thus, as Haynes, et al., point out [HAYN82], "a very powerful way to optimize [a computer system] is to tailor them to a specific problem or class of problems". By carefully choosing the problems and appropriate algorithms to solve them, it is possible to design systems that exactly satisfy the requirements of the problems. However, the disadvantage of such a design methodology is that the applications that can be programmed on the special system become too restrictive.

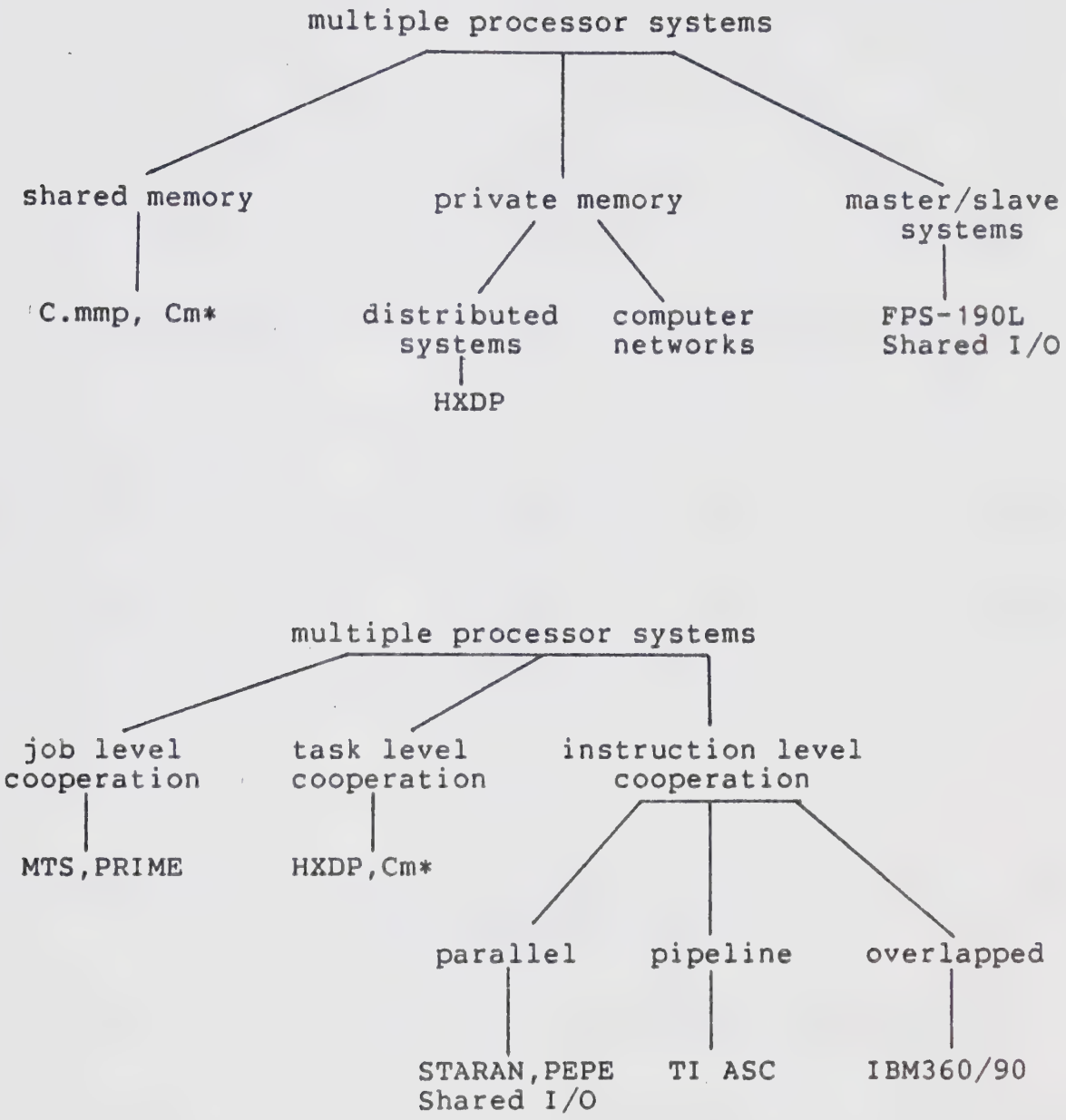


Figure 1.1. Shared I/O in relation to other multiple processor systems

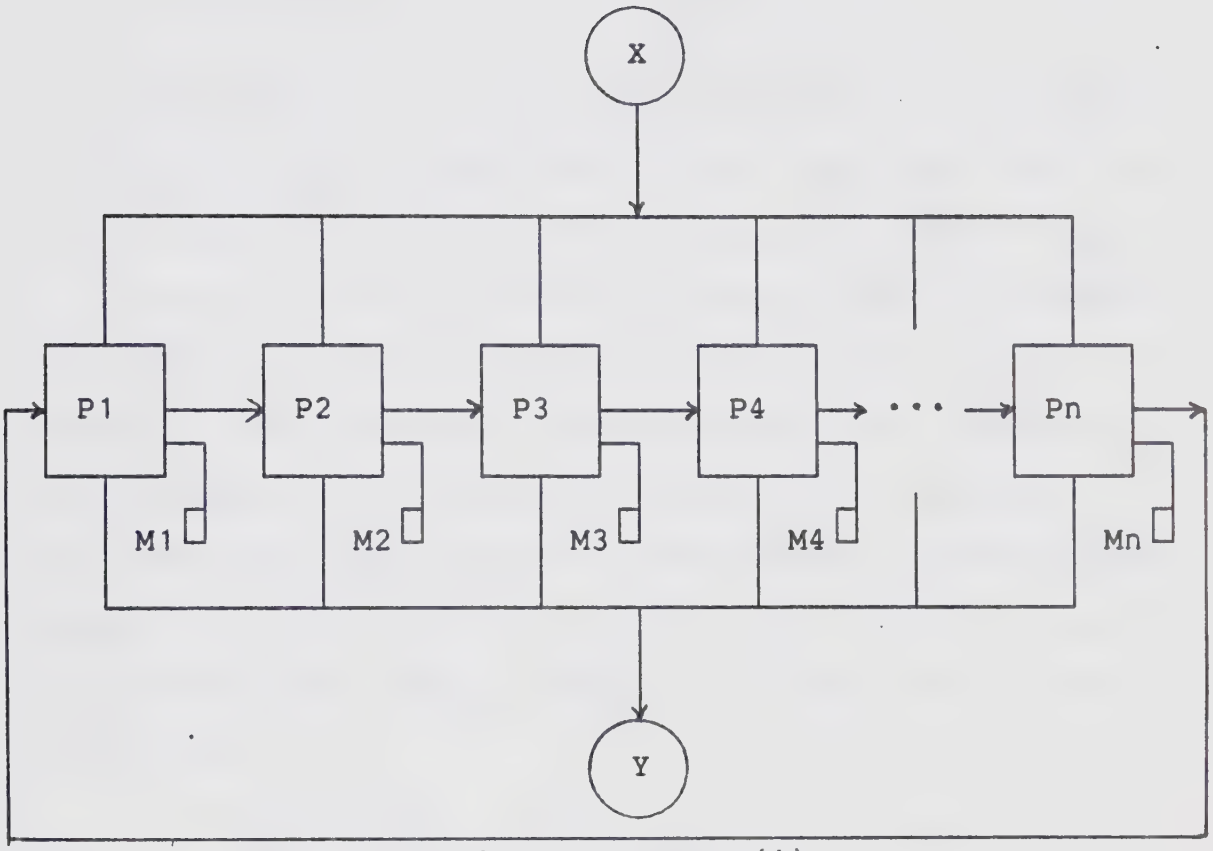


Figure 1.2. n processors in the shared I/O configuration

CHAPTER 2

Shared I/O Organization

2.1. Introduction

The shared I/O organization is designed to implement a certain class of algorithms. These algorithms have the common characteristic that a large amount of time is spent in executing a single main loop. Many signal processing algorithms have such a characteristic. In the shared I/O scheme, all the operations within the loop are assigned to one processor. Each processor executes one repetition of the loop and adjacent processors execute the successive repetitions. Thus all the processors are executing identical program code. As an example, consider the following FORTRAN loop:

```
1         DO 3 I = 1,K
2            Y(I) = A * X(I) + B * Y(I-1) + C * Y(I-2)   2.1
3         CONTINUE
```

Statement 2 constitutes the program for each of the processors. Figure 2.1 shows one repetition of the loop as programmed for the shared I/O organization. Each processor accesses the input device to read an input value, X(I)', and applies the operators defined in the loop body

'Here and in the rest of this thesis, for a variable Z, Z(I) stands for the Ith value of Z, rather than the Ith element of an array Z.

to X(I). This results in the generation of an output value, Y(I), which is then sent to the output device. For the above program, it can be seen that each processor requires two previous values of Y produced by processor(i-1) and processor(i-2) respectively to generate Y(I). In figure 2.1, Y(I-1) and Y(I-2) appearing on the left are the values of Y produced by the repetitions (I-1) and (I-2), respectively.

Since the variable X is used for internal purposes only - that is it is not required by any other processor - it is considered local to the processor producing it. It is referred to as a local variable and each of its values as a local value. On the other hand, the Y values are used in some other repetitions as well as repetition I and consequently are referred to as non-local values. The corresponding variable, Y, is referred to as the non-local variable. Associated with each non-local variable there is a non-local segment which, in repetition I, calculates the non-local value, Y(I). For the local values, there are corresponding local segments which produce them.

2.2. Specifics of Shared I/O

The time lapse between the receipt of a non-local value, say Y(I-1), and the transmission of a corresponding value, Y(I), is denoted by Tns (see figure 2.1). Tns is the delay from the time a non-local value is required from

a previous repetition to the time the corresponding value is available to the next repetition. It is essentially the time necessary to execute the non-local segment associated with that variable.

Tns* is defined as the maximum of all such Tns:

$$Tns* = \max \{Tns\} \qquad 2.2$$

It represents the worst-case time delay between the time when repetition I must receive a non-local value until repetition (I+1) is able to receive the corresponding non-local value. Since the rest of the program on all the processors is identical, Tns* is the time delay between generation of successive output values. Therefore, the maximum throughput that can achieved for a given algorithm, R*, expressed as the number of outputs per unit time is,

$$R* = 1 / Tns* \qquad 2.3$$

Tns* is an important characteristic of an algorithm since it is a measure of the extent to which successive repetitions can be overlapped. Consecutive repetitions have to commence at an interval, Ts, of at least Tns*, so that no processor will require a non-local value before it is produced (figure 2.2).

For certain algorithms Tns* has a value of zero. For example, in the following equation as programmed for the

shared I/O, Tns* is zero (figure 2.3).

$$Y(I) = A \times X(I) + B \times X(I-1) + C \times X(I-2) + D \times X(I-3) \quad 2.4$$

A zero value for Tns* means that if all input values are available before execution begins, then all repetitions of the loop can begin executing in parallel. However, if successive input values are available only after a time delay of Ts as shown in figure 2.4, then the performance of these algorithms is limited by Ts rather than being infinite as defined by equation 2.3.

Let Trep denote the time required to execute all the operations associated with one repetition of the loop. Then the number of processors needed to achieve maximum throughput is,

$$m^* = \lceil (Trep / Tns^*) \rceil \quad 2.5$$

where $\lceil x \rceil$ is the smallest integer greater than or equal to x. That is, if m* processors are available, then the first processor, P(1), completes the execution of its repetition at the same time when the processor adjacent to P(m*) is expected to begin execution. Hence, P(1) can be considered adjacent to P(m*) and begin the subsequent repetition.

The time each processor idles at the end of an repetition, Ti, will be a finite quantity since m* is an integer. This is essentially the time spent on

synchronizing with the input device.

$$Ti = m^* \times Tns^* - Trep \qquad 2.6$$

If less than m* processors are available, then a smaller repetition rate can be accommodated. Combining equations 2.3 and 2.5, we have the following equation, expressing the reduced throughput, R(m), in terms of the number of processors, m,

$$R(m) = m \,/\, Trep \qquad 1 \leq m \leq m^* \qquad 2.7$$

Equation 2.7 indicates that the throughput of the multiprocessor can be varied to achieve a desired level of performance. For example, if the input values occur in real time at an interval of Ts, then the number of processor required is,

$$m = \lceil (Trep \,/\, Ts) \rceil \qquad Ts \geq Tns^* \qquad 2.8$$

If during Trep, P(i) sends n non-local values to P(i+1) before the later receives the first of these n values, then the data transfer is said to be interleaved to a degree n. If P(i) produces a maximum of k non-local values, k ≥ n, then if P(i) sends all of the k values before P(i+1) begins to receive them, then the degree of interleaving is k. In this case, a first-in, first-out buffer of size k, is necessary between the processors to store the values until P(i+1) is ready to receive them. The following discussion applies to the cases where n < k

where we need a buffer of size n.

Figure 2.5 illustrates a hypothetical case in which the degree of interleaving is two. Let Trs denote the elapsed time between a receive executed by processor(i) and the nth send to processor(i+1) following this receive (see figure 2.5). The time processor(i) must wait at the instant of (n+1)th send is (figure 2.6).

$$Tw = Ts - Trs \qquad 2.9$$

The accumulated wait time at the end of each repetition will be,

$$Tw = \sum_{i=2}^{k/n} Ts-Trs \qquad 2.10$$

In order to circumvent this wait time, the minimum of all such Trs in a given program, Trs*, must be such that,

$$Trs* \geq Ts \qquad 2.11$$

(see figure 2.5). For example, consider the equation 2.4, which is a four term finite impulse response filter[2].

[2]A digital filter can be expressed as,
$Y(n) = A_0 \times X(n) + A_1 \times X(n-1) + A_2 \times X(n-2) + \ldots + A_j \times X(n-j) + B_1 \times Y(n-1) + B_2 \times Y(n-2) + \ldots + B_k \times Y(n-k)$
for some integers j and k. A_j and B_k are constant coefficients characteristic of the filter. X(n) and Y(n) are the values of nth input and output respectively. If the response of the filter to an impulse input is of finite duration then the filter is called *finite impulse response* filter, FIR, otherwise it is an *infinite impulse response* filter, IIR. Simply stated, when k is zero in the above relation, we have a FIR. For a IIR, k must be greater than zero. j is three and k is zero in equation 2.4 and for the program 2.1, j is zero and k is two.

Figure 2.3 displays the program that would be executed by each processor. Send and Receive statements identify the transfer of non-local values between processors. In this example, the degree of interleaving is one.

Processor P(0) initiates the execution by sending the initial values of X1, X2 and X3 to its neighbor processor P(1). P(1) reads the first input value from the input device and produces the corresponding output value Y in accordance with equation 2.4. In the course of the execution, it sends a set of three new non-local values to processor P(2). Each processor performs identical operations on successive input values to produce successive output values. Figure 2.7 shows an instance of a particular implementation using six processors (only P(1), P(2) and P(3) are shown). For the purpose of illustration, in figure 2.7, Trep, Ts and Trs* are assumed to be 51, 9 and 10 time units respectively.

Figure 2.8 shows a case when Trs* is zero and Ts is 9 time units. Since the number of sends in a repetition is three, by equation 2.10, the total wait time on send is,

$$Tw = (9-0) + (9-0)$$
$$= 18 \text{ time units}$$

This increases the Trep by the same amount. The number of processors required then equals,

$$m = \lceil (51+18)/9 \rceil$$

$$= 8$$

In figure 2.7, the send instructions are moved down until Trs* becomes greater than Ts. The wait time on send can be seen to be zero. Figure 2.9 illustrates a case in which Ts has been increased to such an extent (Ts=12) that the adjustment in Trs* (Trs*=10) resulting from the shifting of the send instructions does not still satisfy equation 2.11. Subsequently the total delay incurred is,

$$Tw = (12-10) + (12-10)$$

$$= 4 \text{ time units}$$

The number of processors required now is,

$$m = \lceil (51+4)/12 \rceil$$

$$= 5$$

2.3. Summary

In this chapter we presented the details of the shared I/O organization. It is a special purpose organization developed for implementing a few specific algorithms. Many of the signal processing algorithms, like the FIR and IIR filters we considered here, have been found appropriate for implementation on the shared I/O system [HEUF80]. It operates by overlapping successive repetitions of a program loop to an extent determined by the characteristics of the algorithm. We developed relations that described the throughput, the number of the processors,

the idle time on input synchronization of the system and certain conditions to reduce the delays incurred during transfer synchronization. In the next chapter we will present a comparison between shared I/O and the concept of pipelining.

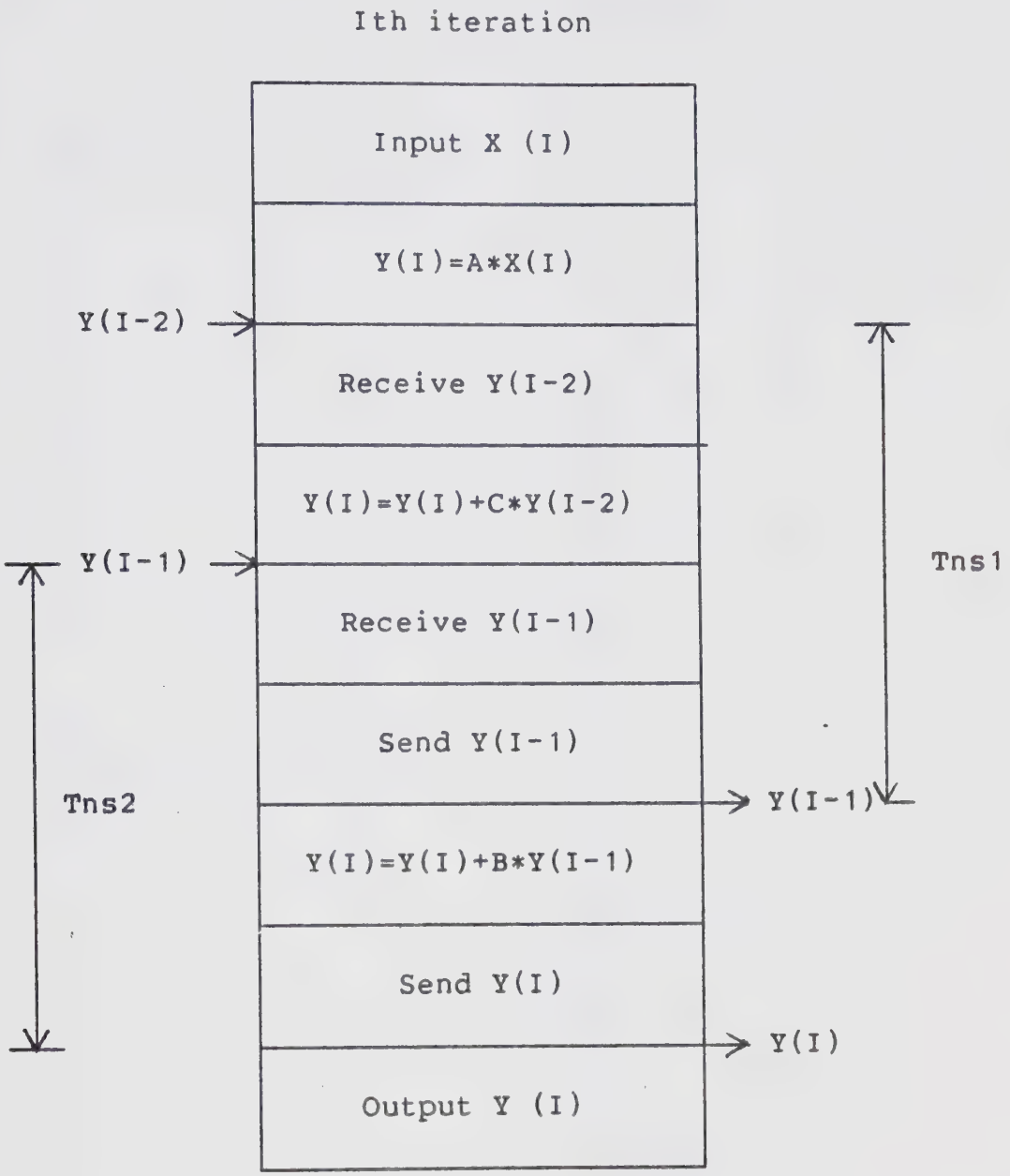


Figure 2.1. Ith iteration of program 2.1

Tns = max (Tns1, Tns2) > 0

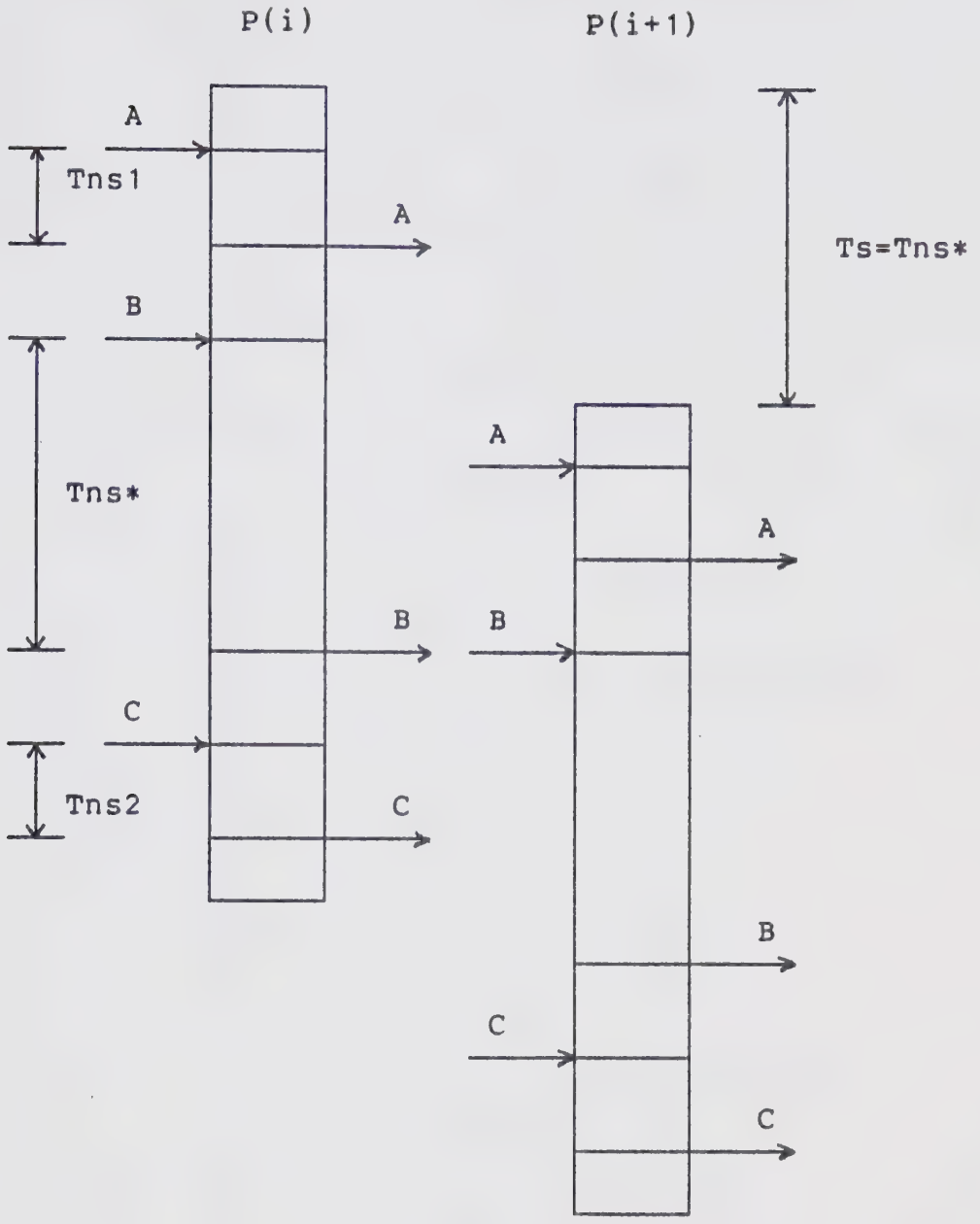


Figure 2.2. Scheduling adjacent processors with Tns* > 0, Ts = Tns*

A,B and C are non-local values. If the interval, Ts is less than Tns*, then P(i+1) would have waited at second receive until B becomes available.

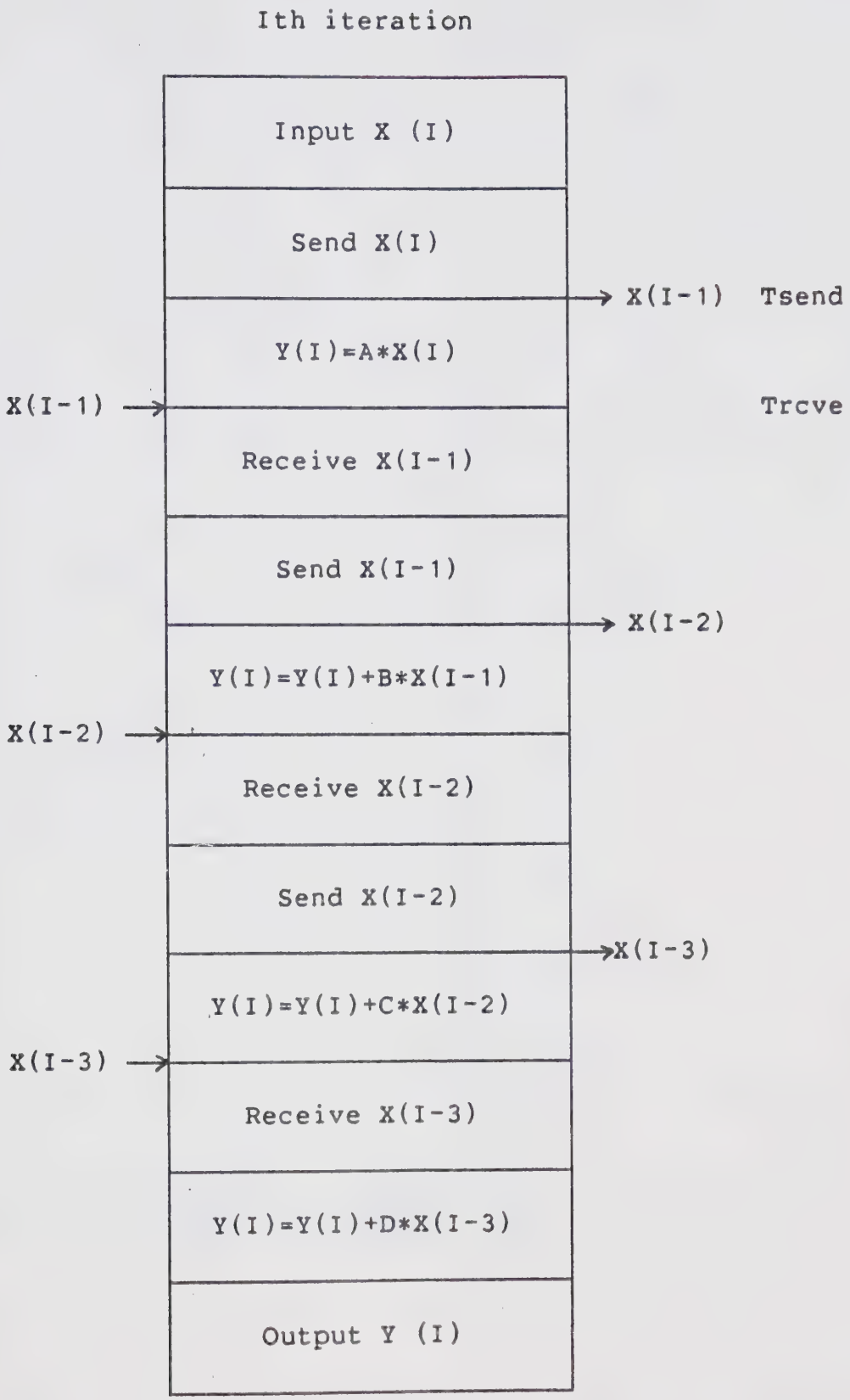


Figure 2.3. Ith iteration of equation 2.4
Tns* = Trcve-Tsend = 0 (since time cannot be negtive)

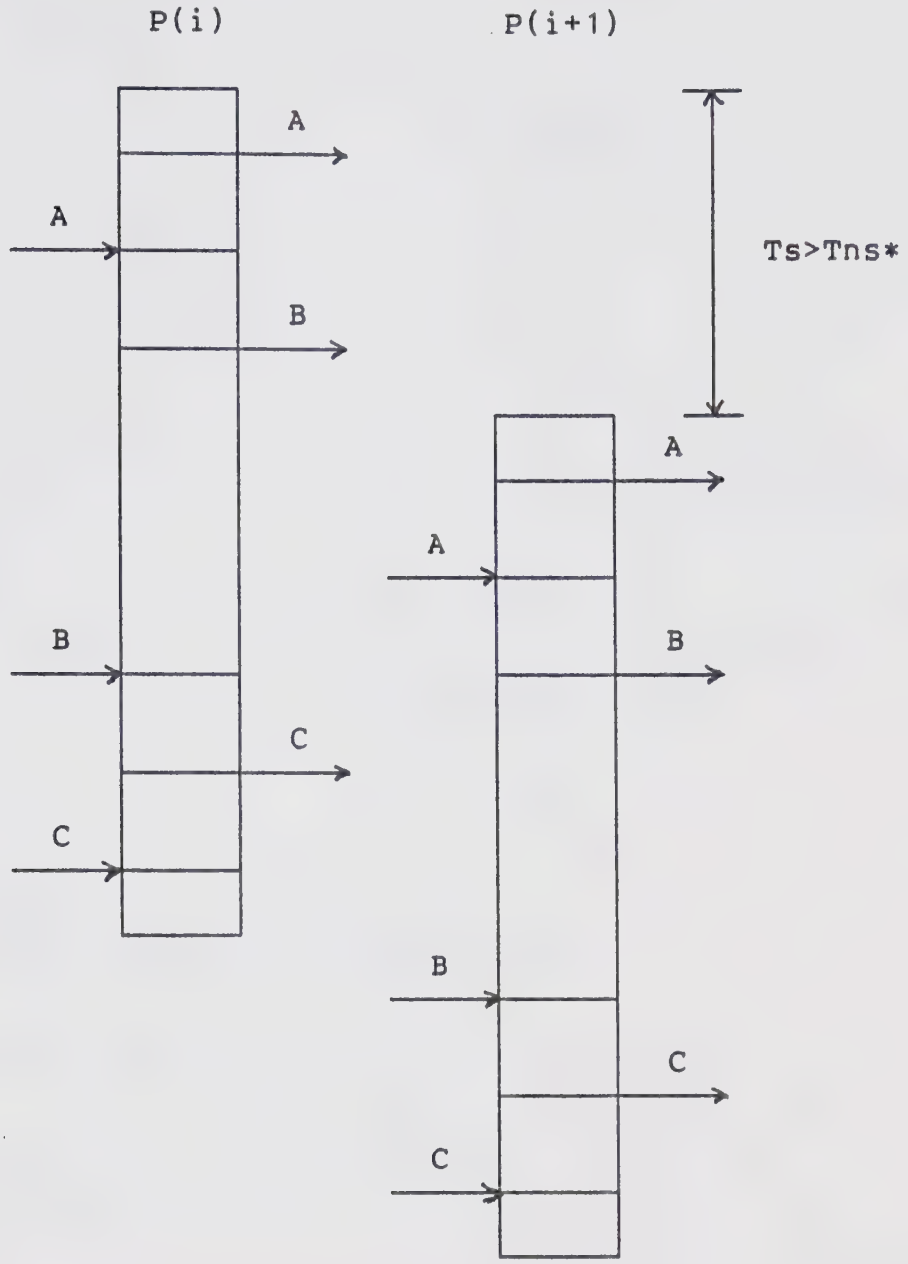
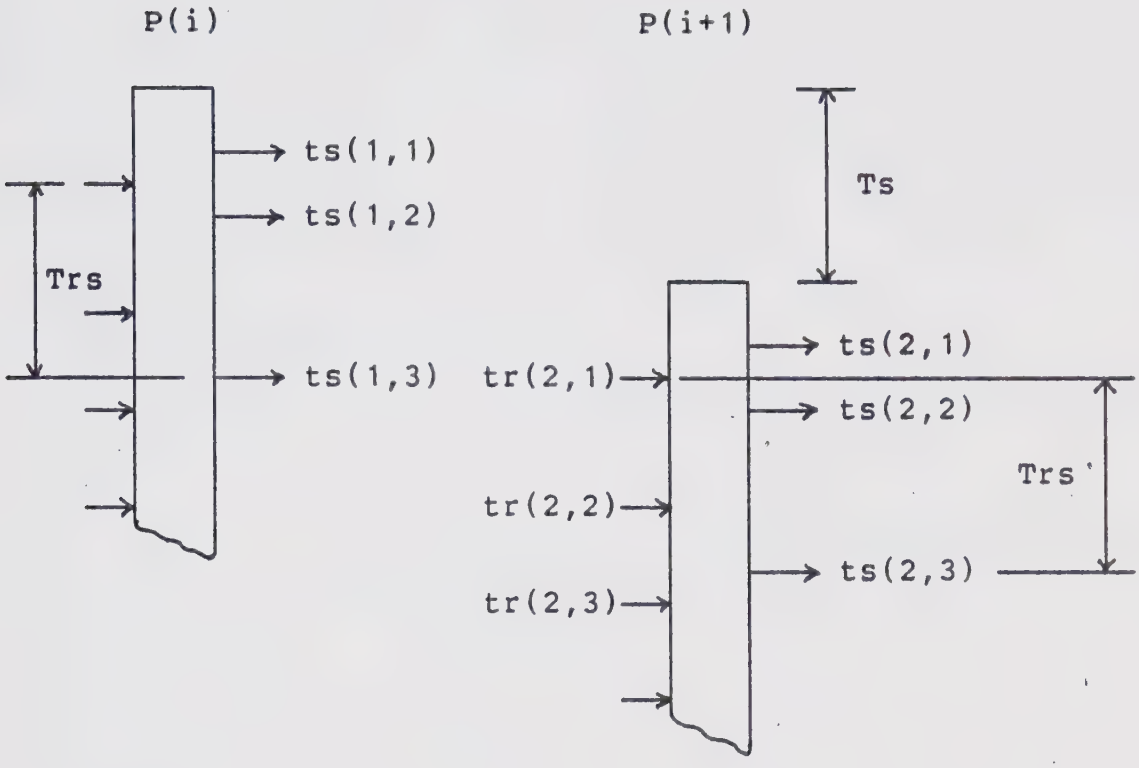


Figure 2.4. Scheduling adjacent processors with Tns* = 0 and Ts > Tns*



Legend:
ts(i,j): time P(i) sends value j
tr(i,j): time P(i) receives value j

For zero wait time at ts(1,3), we have,

$$ts(1,3) \geq tr(2,1)$$
$$tr(2,1) = ts(2,3) - Trs$$

If there is no wait time then,

$$ts(2,3) = ts(1,3) + Ts$$

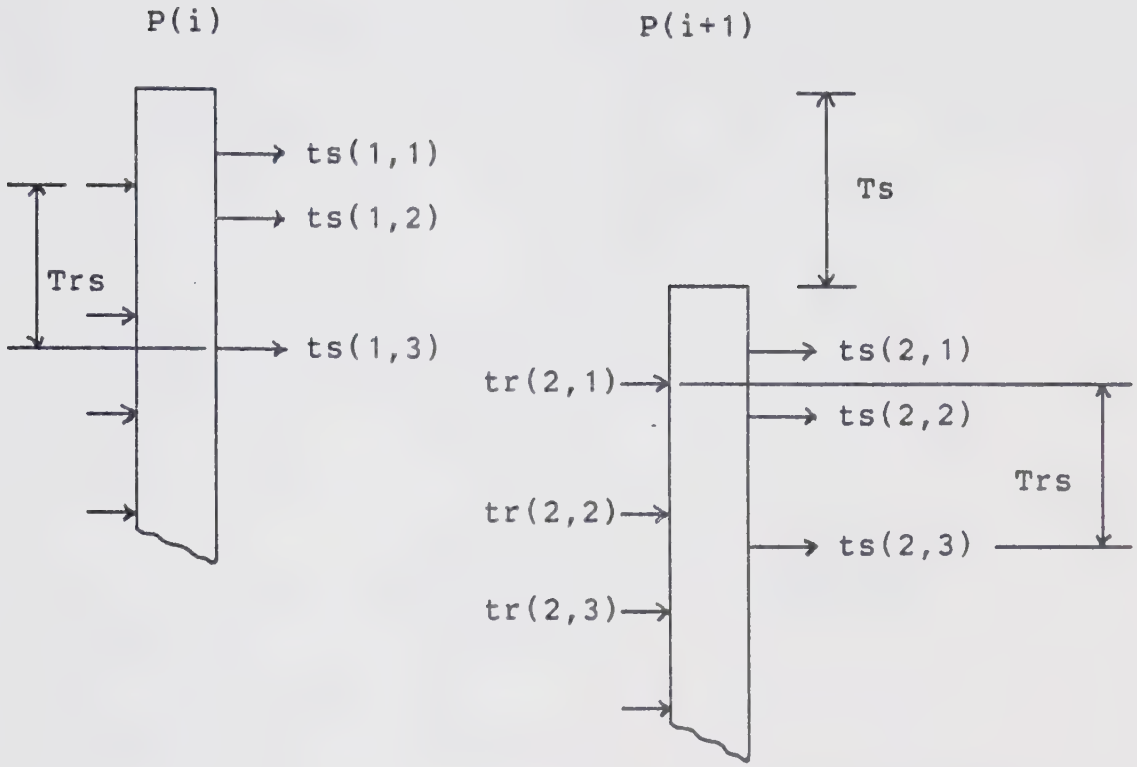
Therefore,

$$ts(1,3) \geq ts(1,3) + Ts - Trs$$

Or,

$$Trs \geq Ts$$

Figure 2.5. Interleaved data transfers with the degree of interleaving as two



Legend:
ts(i,j): time P(i) sends value j
tr(i,j): time P(i) receives value j

At ts(1,3), P(i) must wait since the buffer is full.
The wait time, Tw = tr(2,1) - ts(1,3)
tr(2,1) = ts(2,3) - Trs
ts(2,3) = ts(1,3) + Ts
Therefore,
Tw = Ts - Trs

Figure 2.6. The case when Trs < Ts with the degree of interleaving as two

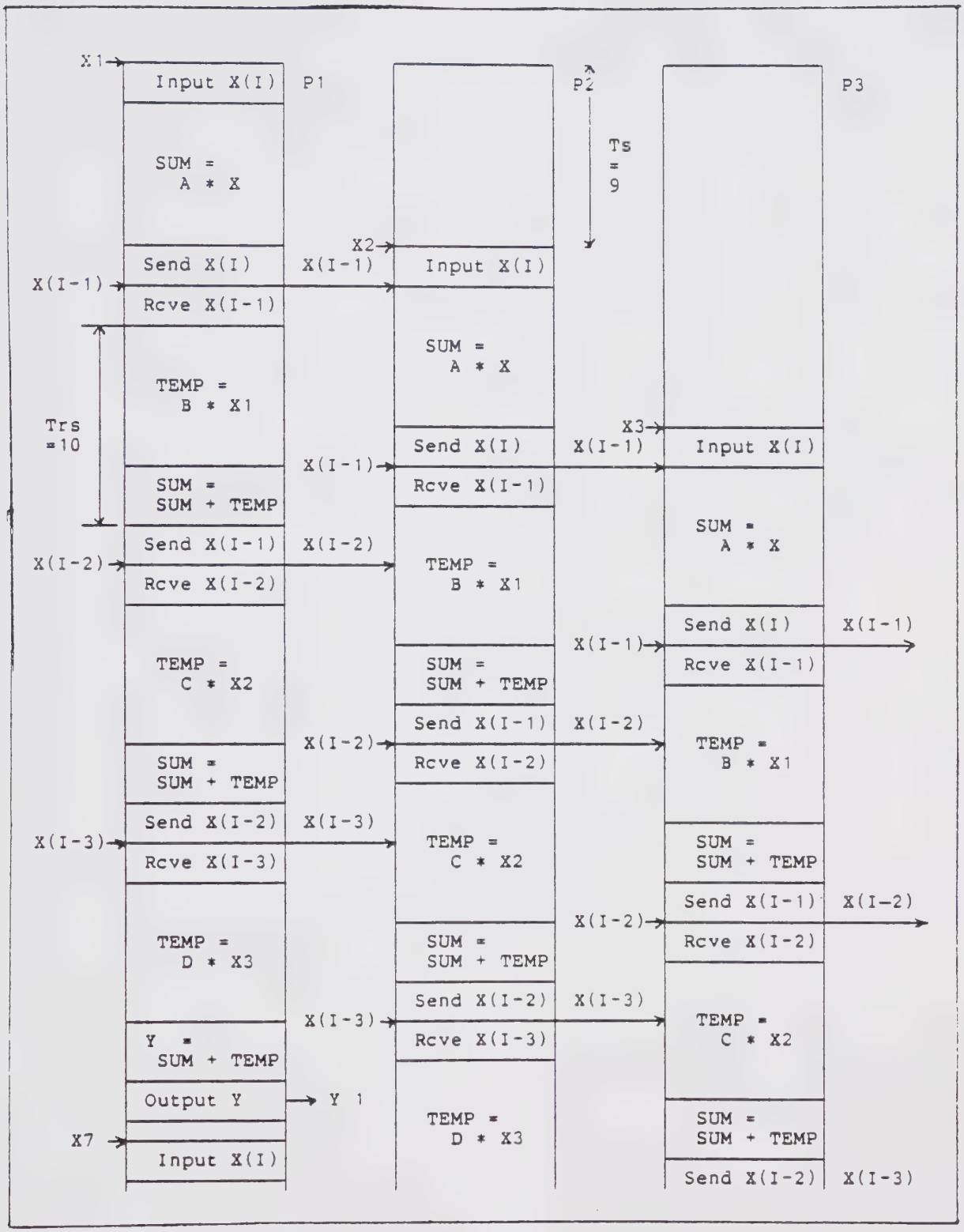


Figure 2.7. An instance of equation 2.4

with Ts=9, Trs=10, Trep=51, m=6.

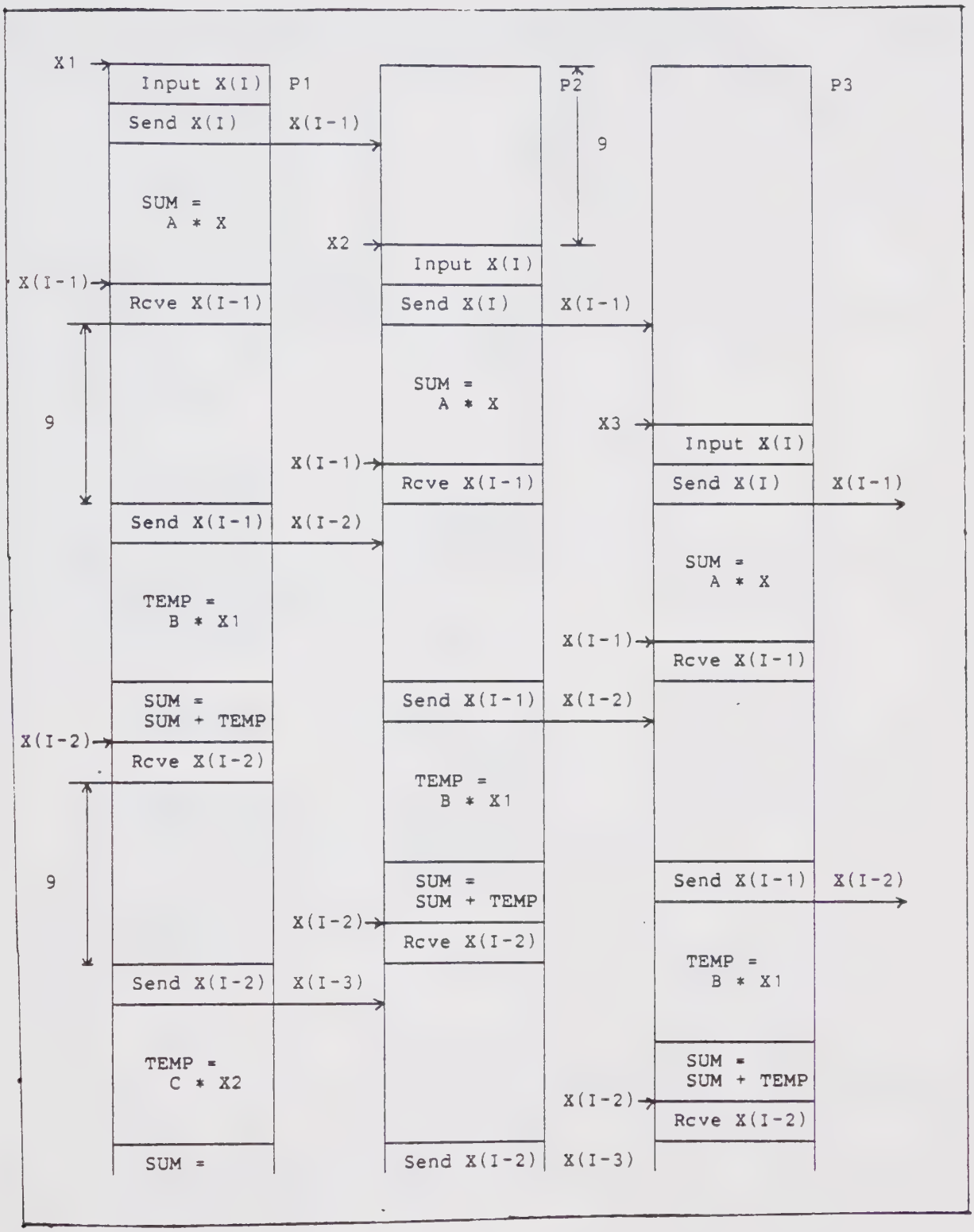


Figure 2.8. Another instance of equation 2.4

with Ts=9, Trs=0, Trep=69, m=8.

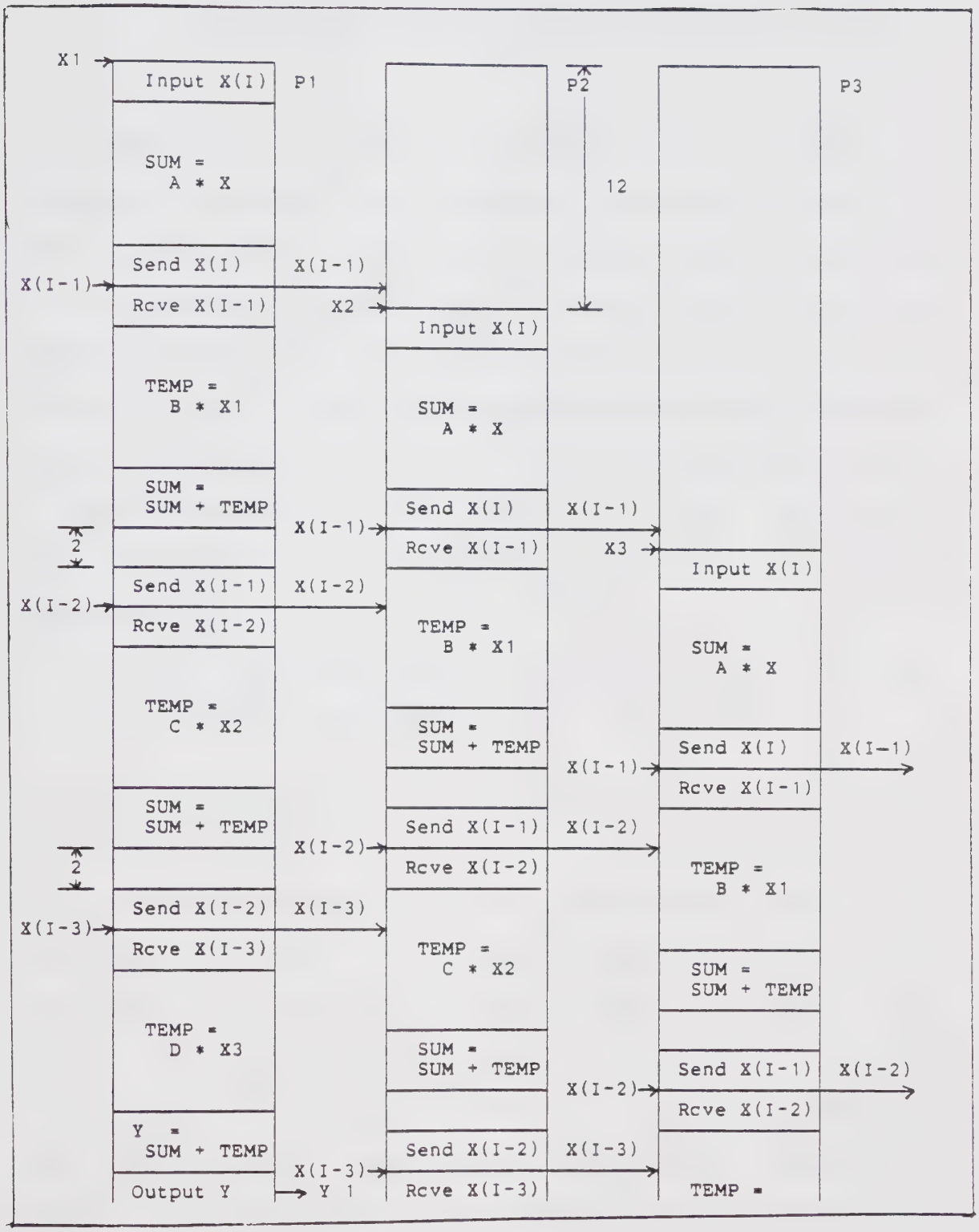


Figure 2.9. Yet another instance of equation 2.4

with Ts=12, Trs=10, Trep=55, m=5.

CHAPTER 3

Comparison with pipeline structures

Techniques to achieve increased speed of operation include parallel and pipelined execution. The first achieves high performance by executing several evaluations of a function in parallel on different data. The later splits the function to be performed into several subfunctions. Each of the divisions is allocated to separate piece of hardware to execute. Both the methods are useful in applications where repeated evaluation of a function is required. As an example, consider the following FORTRAN program:

```
1       DO 3 I=1,N
2          Y(I) = A * X(I) + B * Z(I)          3.1
3       CONTINUE
```

In pipelined execution method, the function represented by statment 2, could be executed in two stages. In the first stage a partial value of Y(I) is generated by multiplying X(I) by A and passed to the second stage. Here the operation B*Z(I) is carried out and added to the partial result. Increased speed of operation is achieved by overlapping successive evaluations of Y(I). In the parallel execution method, several evaluations of statement 2 could be simultaneously carried out on several processors.

Though both methods achieve the same goals, for some applications, there are certain advantages in using one of them in preference to other. A general comparison of these two methods can be found in [KOGG81]. In this chapter, we will restrict ourselves to the comparison of pipelining and shared I/O which performs a computation in parallel. In the next section we will make the comparison between pipelining and shared I/O in terms of throughput and efficiency. As an example, we will consider the implementation of FIR filter in a systolic array. Systolic arrays work on the principle of pipelining. In the later part of the chapter, we will discuss some more details that are specific to shared I/O in comparison with pipeline structures. These details include such aspects as incremental expandability, software and hardware requirements et cetera.

3.1. Performance Considerations

Performance evaluation includes the measurement of throughput of the system and the efficiency expressed in terms of system utilization.

3.1.1. Throughput Considerations

Throughput is defined as the number of outputs or instructions processed per unit time. Consider figure 3.1 which shows a pipeline structure with four functional

units. Each unit implements a particular subfunction. In non-pipelined implementation, the total execution time would be $Tnp=t_1+t_2+t_3+t_4$ time units where t_i for $i=1,4$, is the time to evaluate each subfunction. That is, for every Tnp time units there is an output from the system. In the pipeline version, it would require only Tp units of time to perform the same operation where $Tp=\max\{t_i\}$. The stage of the pipeline that needs $\max\{t_i\}$ time, is referred to as the bottleneck. The maximum performance is limited by the capability of the bottleneck.

For an equivalent implementation in shared I/O, the throughput is defined by equation 2.3 as $1/Tns^*$. If the input rate, 1/Ts, is such that $Trs^* \geq Ts \geq Tns^*$, then the throughput is only limited by 1/Ts rather than the evaluation a particular subfunction; as 1/Ts increases, the throughput will increases proportionately. However, if $Tp \leq Ts$ for some Ts, $Trs^* \geq Ts \geq Tns^*$, then a pipeline structure may achieve a greater throughput than shared I/O.

3.1.2. Efficiency Considerations

Efficiency or utilization factor, relates to the number of subunits of an organization that are busy in a given period of time. In a pipeline, there may be several subunits each requiring different amount of time to evaluate the subfunctions. This complicates the measurement of

efficiency because, some of these subunits may be busy while some other are idle. However, assuming equal execution times (identical ti) for all the subunits, a simple relation may be derived. Lee [LEE 80] defines the utilization factor, Uf, as,

$$Uf = \frac{Tseq}{p \times Tpar}$$

If L is the number of functions to be evaluated, then Tseq is the time to evaluate the L functions on a sequential machine, Tpar is the time to evaluate the same number of functions with p processors. For a pipeline, p is equal to n, the number of stages. Let each function be made up of n subfunctions[1]. If the unit of time is the time to evaluate one subfunction, then,

$$T = L \times n \text{ time units}$$

When executed in a pipelined fashion, it requires n time units to evaluate the first function and (L-1) time units to evaluate the rest. Thus,

$$Tpar = n + (L-1)$$

Then the utilization factor is,

---

[1]L could be the number of loop repetitions to be executed and the function could be the loop body. For example, in program 3.1, N is L and statement 2 is the function to be evaluated. This function is made up of two subfunctions, namely, A*X(I) and B*Z(I) (n is 2).

$$Uf = \frac{L \times n}{n \times \{n + (L-1)\}}$$

Or,

$$Uf = \frac{L}{n + (L-1)} \qquad 3.2$$

It can be easily deduced that Uf is always less than unity unless L is infinity or n is one. When n is one there is no pipelining. When L is infinity, the computation has lasted long enough to offset the initial delay involved in filling up the pipe.

For the shared I/O, the time to evaluate one complete function is Trep. Then the time, Tseq, to evaluate L functions sequentially is,

$$Tseq = L \times Trep$$

It takes Trep time units to obtain the first result and there is an output from the system every Ts time units thereafter (recall that the throughput for shared I/O is 1/Ts). Then Tpar is,

$$Tpar = Trep + (L-1) \times Ts$$

Thus the utilization factor for shared I/O is,

$$Uf = \frac{L \times Trep}{m \times \{Trep + (L-1) \times Ts\}}$$

Further simplification will arise if we assume that m = (Trep/Ts). We get,

$$Uf = \frac{L}{m + (L-1)} \qquad 3.3$$

We note that this relation is similar to equation 3.2. An utilization factor of one is attainable only when m is one or L is infinity. That is, both pipeline and shared I/O utilize the available resources to the same degree.

Thus, we have shown that shared I/O performs better than pipeline in terms of throughput for certain algorithms (for which Ts < Tp). Since the degree of utilization is same for both approaches, the overall performance of shared I/O is better than pipeline for these algorithms.

In the next section, we will consider an example of a pipeline implementation of FIR algorithm and compare the results with that obtained for a similar implementation on shared I/O. Our aim here is to illustrate the differences in performance between shared I/O and a pipeline that has actually been implemented.

3.2. Systolic Arrays: an example

Systolic arrays are highly regular structures that function on the principles of pipeline. A systolic system consists of a set of interconnected cells, each capable of

performing some simple operation. Intermediate results from a computation flow through the cells in a pipelined fashion [KUNG82]. Originally designed for doing matrix computations efficiently and cost-effectively with VLSI technology, it can be easily be adapted for signal processing applications by formulating the problems as matrix operations. Kung and Leiserson describe a method of performing the necessary calculations for the FIR filter [KUNG80].

A linearly connected network is employed in which the inputs to each cell are from the left, right and the top. Figure 3.2 shows one cell of the systolic array and one processor of shared I/O configured to implement the FIR filter. Recalling equation 2.4[2], we notice that in shared I/O, the coefficients of each term are stored within the processor during power-on time, and stay until the completion of the computation. The X values move from one processor to another after entering from the top, and the Y values stay only to be output at the end of one computation. In the systolic cell, the coefficients move in from the top, a fresh set entering the cells for each computation, the Y values move from right to left and the X to the right from left. Figure 3.3 shows the FIR filter expressed as a multiplication of the coefficient matrix,

[2] Equation 2.4:
$Y(I) = A \times X(I) + B \times X(I-1) + C \times X(I-2) + D \times X(I-3)$.

C, with the X matrix producing the Y matrix as the result. Figure 3.4 shows the cell connections to perform the desired multiplication. During the course of the multiplication, only alternate cells are active on each cycle. Each cell performs one multiplication and one addition, producing,

$$y^{k}(i) = y^{k-1}(i) + C(i,k) \times x(k)$$

where, $y^{k}(i)$ is the kth approximation to ith value of y, C(i,k) is the (i,k)th element of coefficient matrix C and x(k) is the kth element of x. Thus the cycle time is defined as the time to do one multiplication and one addition. This represents the time required by each stage of the pipeline to execute the multiplication and addition operations. Figure 3.5[3] shows the first few steps of the multiplication. If there are w terms in the in the FIR filter (in equation 2.4, w is four) then w cells are required to implement the algorithm. Then the total time required to complete the multiplication is given by,

$$T = 2(n-1) + w \quad \text{time units}$$

Or,

$$T = 2(n-1) + 4 \quad \text{time units} \qquad 3.4$$

After the first w time units, the Y values begin to emerge on the left side and every two cycles we have another Y

[3] In this figure, $y^{k}(i)$ is simply denoted as yi and x(k) as xk.

value. If n is number of elements in Y then we need another 2(n-1) cycles to compute the remainder of Y, which explains the above equations.

In comparison, for the shared I/O, we need 12 time units to generate the first Y value (4 time units for performing 4 multiplications and 4 additions[4] and 8 time units for 8 transfers[5]) and Ts time units to evaluate successive values of Y, totally requiring,

$$T = (n-1)Ts + 12 \text{ time units.} \qquad 3.5$$

Comparing the first terms of equations 3.4 and 3.5, we observe that for the systolic array implementation, the throughput is limited by the time required to do one multiplication and one addition. In the shared I/O, it varies as a function of Ts. The throughput is more in the shared I/O when Ts is less than 2 time units. Ignoring the constant terms, the overall execution time is also less in shared I/O when Ts is smaller than 2 time units.

3.3. Other Aspects

In this section, we will compare the pipeline and shared I/O organizations on a more general term. We have

[4]There are, in fact, only 3 additions which means it requires less than 4 time units to generate the first Y value.

[5]Assuming each transfer would take as long as one multiplication and one addition, which is a rather conservative estimate.

already discussed the performances of both the organizations in terms of throughput and utilization. Here, the comparison is in terms of data movement, hardware, software requirements, incremental expandability and testing.

Movement of data:

The local values in a program stay within a processor in the shared I/O and the non-local values move between the processors. In a pipeline implementation, they may either stay within a stage or move between the stages. For example, in the systolic array implementation of FIR filter, X and Y values move from stage to stage. In the shared I/O, Y values stay within the processor and X values move between the processors. That is, the stages in a pipeline may either implement a local segment of the shared I/O or a non-local segment. If Tls* is the maximum time to evaluate one local segment, then the throughput of the pipeline is limited by max(Tls*,Tns*). For the shared I/O, the limit on the throughput is Tns*. Thus, shared I/O performs better when Tns* < Tls*.

Hardware:

For each stage in the pipeline, the minimum amount of hardware required includes the hardware necessary to implement the subfunction of the stage and two connections to establish communication with neighbor

stages. For the shared I/O, each processor should be able to implement the entire function, and should have two pairs of connections: one pair for communication to the input and output devices and the other for transferring non-local values. For the pipeline, at the very least the first and last stages are different from the rest of the pipeline which have to interface to the external input, output devices. The processors of the shared I/O, on the other hand, are all identical including their external connections.

Software:

The software for the shared I/O is identical for all the processors except for the first processor which contains the initialization code. The first processor initializes all the non-local variables to their pre-determined values and begins the execution of the function by transferring them to the second processor. For the pipeline, depending upon the subfunction implemented by each stage, it is different for all the stages.

Incremental expandability:

Once a function has been partitioned and the subfunctions have been allocated to the various stages it is difficult to expand the pipeline by adding another stage. The expansion would necessitate modifying the subfunctions implemented by all other stages.

However, with the shared I/O configuration, another processor can be easily added to the structure by reconfiguring the inter-processor communication links between the two processors where the new unit is to be attached.

Testing

From equation 2.7[6], we note that, the shared I/O can be operated with only one processor. In this minimum configuration, a processor is considered adjacent to itself. The inter-processor connections of the processor is connected to itself. Operating at the lowest throughput, this feature enables the testing and debugging of the programs to be executed by the processors, using only one processor. At the completion of the testing, sufficient processors can be added to achieve the desired performance level. With the pipeline organization, this is not possible.

Table 3.1 summarizes the above discussion. The last entry in the table refers to the need to decompose a loop body in the case of a pipeline, so that the partitions are executed on separate stages. For the shared I/O, such a division is not necessary because the entire loop body is executed by each processor. Both the approaches have some programming overheads in the form of detecting

[6] Equation 2.7: $R(m) = m/Trep$.

dependencies between various statements within a loop. These dependencies have to be strictly resolved in order to produce correct results. Analyzing a program for data dependencies is not a trivial task [HEUF80].

3.4. Summary

We have compared the shared I/O organization to the pipeline structures and have discussed a number of features that make the shared I/O organization attractive for certain applications. We also identified that the pipeline is more effective when Tp is such that $Tp < Ts$, $Trs* \geq Ts \geq Tns*$. We should mention here that we made a basic assumption about the operations within a loop for implementation on either shared I/O or pipeline. We assumed that all the loop repetitions are identical. In some cases this may not be true. For example, if the loop body contains conditional statements, then depending on the conditions prevailing at execution time, some operations of the loop may not be executed. For such cases, neither the pipeline nor shared I/O achieves optimum performance.

	Pipeline	Shared I/O
1) Values transferred	local or non-local or both	non-local
2) Performance limited by	max(Tls*, Tns*)	Tns*
3) Hardware	different for each stage with one input and output connections for each stage. At least the first and last stages are different	same for all processors with four connections required for each: 2 for input and two for output
4) Software	different for all stages	identical for all processors[1]
5) Incremental expandability	not possible	can be done with ease
6) Prototype using only one processor	no	yes
7) Decomposition of the loop body	yes	no

[1] Except for the first processor which contains the initialization code.

Table 3.1 Comparison of shared I/O and pipeline

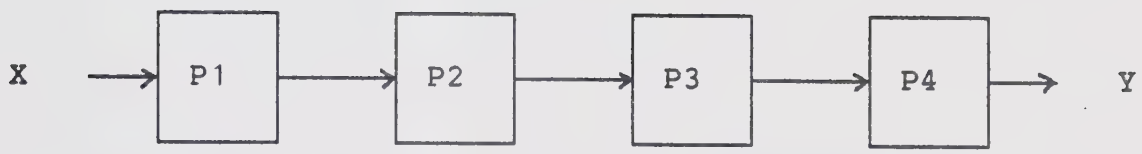
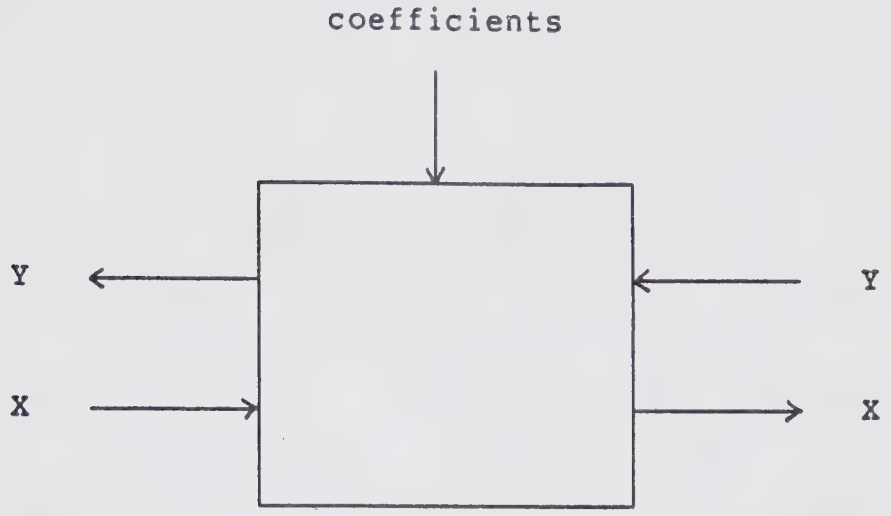
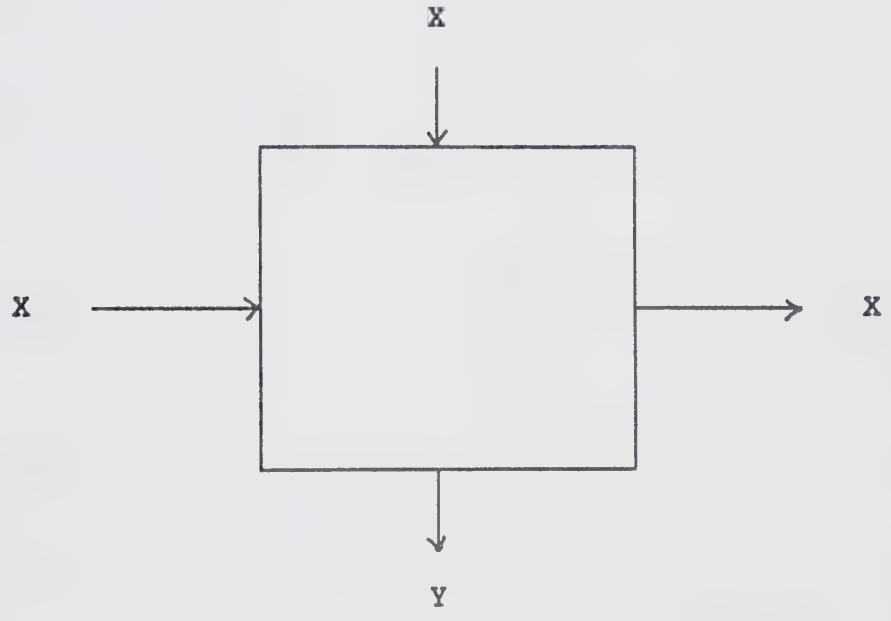


Figure 3.1. 4 processors in pipeline configuration



One cell of the systoloic array for the FIR computation



One processor of the shared I/O for the FIR computation

Figure 3.2. Systolic cell and shared I/O processor for the FIR filter

$$\begin{bmatrix} A & B & C & D & & 0 \\ & A & B & C & D & \\ & & A & B & C & D \\ & & & A & B & C \\ 0 & & & \cdot & & \\ & & & \cdot & & \\ & & & \cdot & & \end{bmatrix} \begin{bmatrix} x1 \\ x2 \\ x3 \\ \cdot \\ \cdot \\ \cdot \\ xn \end{bmatrix} = \begin{bmatrix} y1 \\ y2 \\ y2 \\ \cdot \\ \cdot \\ \cdot \\ yn \end{bmatrix}$$

$$C \quad X = Y$$

Figure 3.3. FIR filter expressed as a matrix multiplication

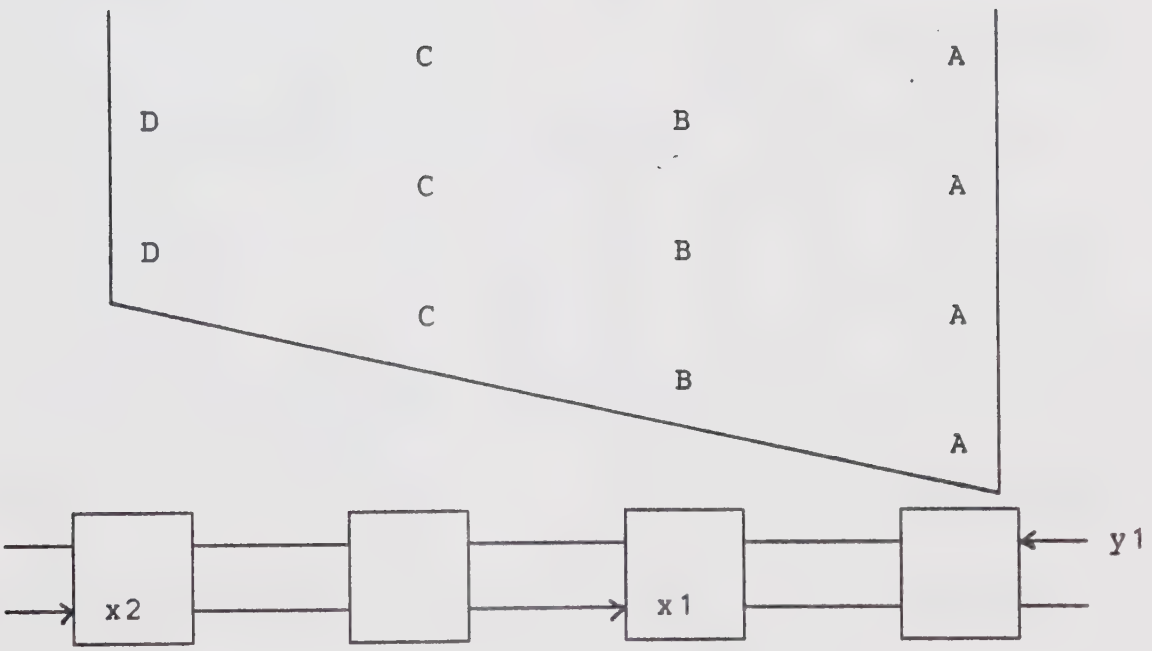
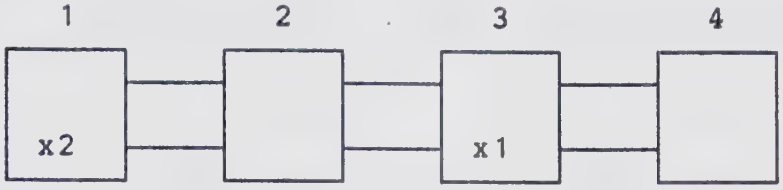
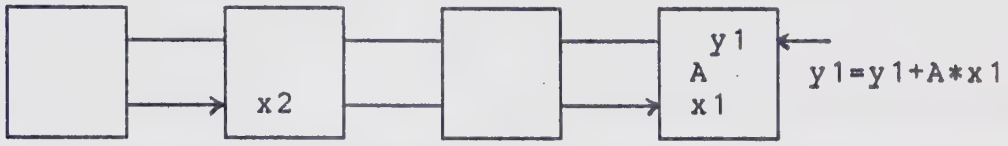


Figure 3.4. Systolic cell arrangement for FIR filter

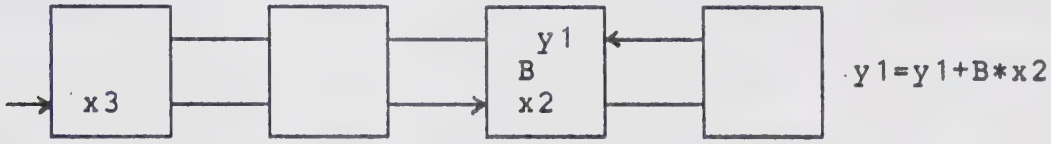
Step 0. During the first three steps, x1 is moved to P(3) and x2 to 1 as part of the initialization. Y is initialized to zero.



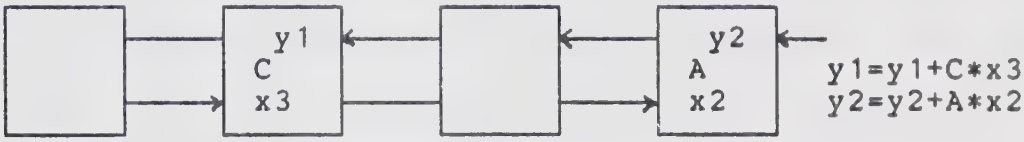
Step 1. y1 enters 4. A enters 4. x1 and x2 move right to 4 and 2.



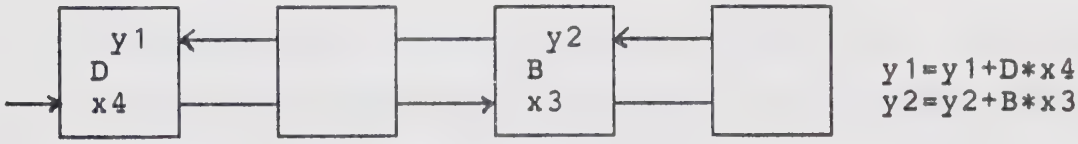
Step 2. y1 moves left to 3. B enters 3. x2 moves right to 3. x3 enters 1.



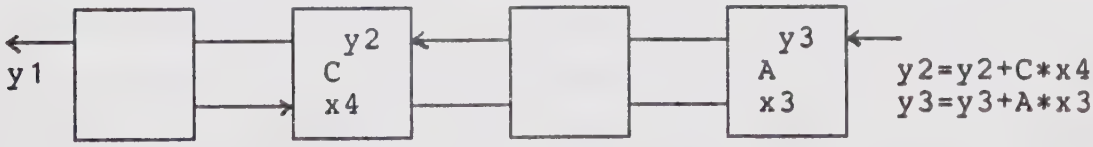
Step 3. y1 moves left to 2. C enters 2. A enters 4. y2 enters 4. x2 moves right to 4 and x3 to 2.



Step 4. y1 moves left to 1. D enters 1. B enters 3. y2 moves left to 3. x3 moves right to 3. x4 enters 1.



Step 5. y1 is output. C enters 2. A enters 4. y2 moves left to 2. x4 moves right to 2 and x3 to 1. y3 enters 4.



Step 6. y2 moves left to 1. D enters 1. B enters 3. y3 moves left to 3. x4 moves right to 3. x5 enters 1.

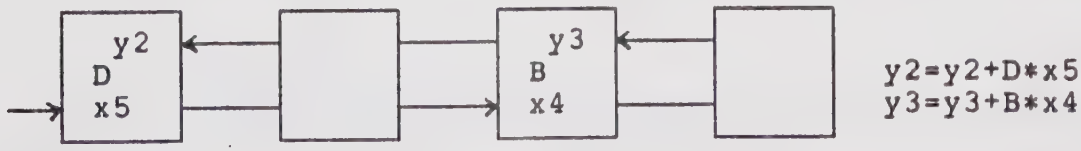


Figure 3.5. First seven steps of the FIR algorithm

CHAPTER 4

Inter-Processor Communication Mechanisms

An important consideration in the design of shared I/O is that data transfers between the processors be completed in as short a time span as possible. Since the sends and receives occur on a time scale comparable to instruction execution time, if a large percentage of the processing of a loop repetition is spent in supervising the data transfers, the performance gained by the parallel execution of the loop repetitions may be lost. Also, Tns* includes the time expended in the data transfers (see figure 2.2). Thus by equation 2.3[1] any reduction in the time required for the transfers would directly result in the enhancement of the throughput.

The overheads involved in data transfers are in the form of observing tight synchronization between the sender and receiver. In the shared I/O, processor(i) sends values to processor(i+1) and receives from processor(i-1) (see figure 1.2). To allow proper operation, it must be ensured that P(i) does not send a second value to P(i+1) before the later has read the first[2]. Similarly, P(i)

---

[1] Equation 2.3: R=1/Tns*.

[2] It is assumed that there exists a buffer of size one between the processors, P(i) and P(i+1)

must be prevented from receiving from P(i-1) when there is nothing to receive. This could be managed in software by incorporating a busy loop in the program. Once within the loop P(i) would repeatedly access the status of its neighbors and act accordingly: either wait or go. This introduces an unnecessary processing overhead as a result of having to execute the busy loop even when the circumstances are such that P(i) does not have to wait. If implemented in hardware, such a loss of precious time can be eliminated. That is, we could dispense with the delay of sending if the previous value has already been received and the delay of receiving if the next value has already been sent. In this chapter, we describe two hardware designs to handle the inter-processor communication as required.

In the first, a hardware unit maintains a buffer size one to hold the data sent by P(i). The buffer is considered full when P(i) sends a value to it and empty when P(i+1) reads from it. The buffer full, empty conditions are termed as exceptional conditions. P(i) is forced into a wait state[3] if it tries to send a data when the buffer is full. It is released from the wait state when the buffer becomes empty. Similarly P(i+1) is put in a wait state when it tries to read from an empty buffer and

[3] In a wait state, the processor waits and does nothing.

released from it when the buffer turns full. Figure 4.1 illustrates the above protocol. Note that this requires no knowledge on the part of the software, about the availability of the buffer for a data transfer.

The second design is a more general version of the first where there is a buffer of size more than one. If the size is n, then it permits at the most n sends from P(i) to take place before any receives from P(i+1). Once the buffer is full, it allows n receives from P(i+1) to take place before any send from P(i). In the following section we describe the processor selection. The primary requirement for a processor is that it be capable of entering a wait state as explained above.

4.1. Selection of a Processor

The processor has to be selected in a such a way that it permits analysis of different approaches without unduly increasing the cost factor. It must also be flexible enough to allow examination of different algorithms to see if they are feasible for implementation on the shared I/O. For the later purpose, it is advantageous if the processor is "programmable"; that is, it can be used under software control. A microprocessor easily satisfies these two constraints. The use of a microprocessor not only allows software reconfigurability but also, at the current market prices, gives a smaller cost to benefit ratio.

Depending upon the techniques used to interface to the peripheral devices[4], microprocessors can be classified into three categories: synchronous, asynchronous and semi-isynchronous[5].

A synchronous microprocessor interfaces with devices of matching speed of operation. That is, the data transfers between the CPU and the peripheral take place within a fixed period of time. At the end of the period, the CPU assumes the successful completion of the transfer and proceeds to execute the next instruction. MC6800 from Motorola is an example. Such a processor is not suitable for our purpose because on exceptional conditions, there is no method of indicating to the CPU that the data transfer could not be completed and hence the CPU must wait.

Asynchronous microprocessors monitor the addressed peripheral and automatically enter a wait state if it does not respond within a specific time period. For instance, in MC68000, the peripheral conveys the completion of the data transfer to the CPU through a Data Transfer ACK-nowledge (DTACK) line. As long as this line is not activated by the peripheral, the CPU waits by inserting extra clock cycles in the current machine cycle. This

[4]These include memory as well as I/O devices.

[5]This classification is based on bus communication techniques discussed by Thurber, et al [THUR72].

conforms to our requirement that the processor must be capable of entering a wait state when necessary. However, the one disadvantage is that sufficient logic must be provided to generate the DTACK signal for *all* data transfers including those that do not spring an exceptional condition. This logic would be in the form of a timing device that would activate and deactivate the DTACK signal at proper instances in time.

Semisynchronous microprocessors operate just as synchronous microprocessors as long as the peripherals are faster than or as fast as the CPUs. Nevertheless, they include a provision by which a slow device, if necessary, could force the CPU into a wait state. For example, the WAIT line on the Zilog's Z80 CPU is used for this purpose. When the WAIT line is active, the CPU enters and remains in a wait state. Note that this is in reverse of the operation of DTACK line on MC68000 which when *not* active at a particular instant in time, forces the CPU into a wait state. For our design, it means that the circuitry must include enough logic to generate the WAIT signal *only* on exceptional conditions. That is, the timing device necessary for MC68000 type microprocessor would be absent. Without investigating into more details, let us state that we opted for the Z80 type microprocessor. Later in this chapter, we look into the use of asynchronous microprocessors in shared I/O.

Apart from the Z80, Intel's INTEL8080A, 8085, MCS6500 from Signetics, CDP1802 by RCA and IM6100 of the Intersil from the 8-bit microprocessor group offer a similar WAIT line facility [OSBO78]. MCS6500 does not allow the use of the WAIT control line during write cycles which is very limiting for our applications. From the 16-bit group INTEL8086, Z8000 and General Instruments' CP1600 and TI9900 from the Texas Instruments have a similar feature [OSBO81]. On CP1600, the maximum length of the wait state is limited to about 40 microseconds by the CPU to keep its internal registers refreshed which indirectly affects the program size. Apart from these two, any other from the above set could be selected as a processor. We used Z80 due to its popularity as a 8-bit microprocessor and its availability in our laboratory.

For the data transfers, we selected the parallel communication mode in which the data is transferred in parallel as a complete entity. This mode of operation is the most suitable for the shared I/O organization because the high bandwidth of the parallel bus allows the transfers to take place in the shortest time [GABL80]. We used the Parallel Input Output (PIO) unit of the Z80 microcomputer family to manage the parallel by word communication.

The following section describes the operation of the PIO. The rest of the chapter deals with the development of the special circuits that control the WAIT line of the Z80

during data transfers. The first, called the HOLD controller, makes use of the registers internal to the PIO to form a buffer of size one to hold the data to be transferred. In the second design, first-in, first-out (FIFO) buffer of size 32 is used to contain the data to be transferred.

4.2. Operation of the PIO

Figure 4.2 shows a box diagram of the Z80 PIO unit as connected to P(i). A and C are the I/O ports[6]. In general, both can function as either input or output port. However, in our design, port A is configured as an input port and port C as an output port. Each port has a register to hold the data to be transferred. Associated with each port, there are two control lines termed as ready (RDY) and strobe (STB). These are also referred to as the handshake signal lines. RDY is an output signal indicating whether the port is available for a transfer of data. It is an active-high signal. STB is an input signal generated by P(i-1) or P(i+1). For port C, STB is used to gate the contents of the port register to the inter-port data bus. For port A, it is used to gate the inter-port data bus to the port register. STB is an active-low signal.

[6]There are two PIO chips on the PIO unit with a total of four ports, A, B, C and D. C and D are the same as A and B except that they are on different chips.

The next section lists the special terms used in the remainder of this chapter. The subsequent section describes the behavior of the PIO during send and receive operations.

4.2.1. Terminology

Special terms introduced in this chapter are explained below. Some of the terms may be found only in the figures.

P(i)	processor i
CPU	Z80 CPU
RD	Read signal from the CPU. When low, it indicates that the the peripheral should place a data in the system data bus.
WR	Write signal from the CPU. When low, it indicates that the system data bus contains a valid data to be output.
IORQ	I/O request signal from the CPU. When low, it indicates that the system address bus (lines A0-A7) contains a valid I/O device address. The RD and WR signals are used in conjunction with IORQ.
WAIT	An input to the CPU which when low forces the CPU insert additional clock periods in the current machine cycle. WAIT(i) is the wait signal for P(i).
A(i), C(i)	Ports A and C of the PIO connected to P(i).

ARDY(i), ASTB(i), CRDY(i), CSTB(i)	These are handshake lines of ports A(i) and C(i) respectively.
CE1	Chip enable 1. selects the PIO chip containing port A.
CE2	Chip enable 2. selects the PIO chip containing port C.
A/B	Selects port A or B within a PIO chip. Low for port A.
C/D	Control or data. Low for data.
T1, T2, T3, T4	Clock periods in a machine cycle used in the timing diagrams. T1 appearing after T3 signifies the start of the next machine cycle.
Tw	Clock period during which the CPU is waiting. for the I/O operations, one Tw is automatically inserted by the CPU.
STD bus	System bus which includes the address lines, data lines and the control lines of the CPU.
PIO bus	Bus that connects ports A and C. this includes the handshake lines.
HOLD bus	Bus that carries the signals generated by the HOLD controller.
FIFO bus	The bus that carries the signals generated by the FIFO circuit.
set	The signal is turned high.
reset	The signal is turned low.
^	Logical AND operator.
v	Logical OR operator.
'	Logical negation.
ms	Milliseconds.
us	Microseconds.

ns	Nanoseconds.

4.2.2. Data transfer operations using PIO

The send and receive operations are implemented using the OUT and IN instructions of the CPU. Figure 4.3 shows the timing relations between various signals during the execution phase of these I/O instructions. During T1, the address of the peripheral becomes stable. For an OUT instruction, the data to be output also becomes stable at about this time. The IORQ and WR or RD are activated in T2. For an IN instruction, the data is read at the falling edge during T3. Just before T1 of the succeeding instruction, IORQ and WR or RD signals are deactivated to signify the completion of the I/O operation. One Tw is automatically inserted by the CPU during these instructions.

4.2.2.1. Send: OUT (nn), A

The OUT instruction when executed by P(i), outputs the contents of the register A to C(i). nn is the address of C(i). Figure 4.4 shows the timing relations between the various PIO signals during the execution of this instruction. CRDY(i) is set by the PIO at the falling edge of the T1 of the succeeding instruction. This denotes that the inter-port data bus contains a valid data for

transfer. CRDY(i) remains high until a raising edge is detected on CSTB(i). At this instant it is reset by the PIO to indicate that there is no valid data in the port output register. The raising edge on the CSTB(i) is generated by the receiver, P(i+1), after it has read the port. If the CRDY(i) is high when the instruction is executed, then it is reset by the PIO at the falling edge during Tw. It is maintained in that state until the falling edge during T1 of the next instruction. This is to ensure that the CRDY(i) is low while the port data is changing. It also ensures that a positive edge is generated on CRDY(i) whenever an output instruction is executed.

4.2.2.2. Receive: IN A,(nn)

The effect of this instruction is to cause the contents of the input port register to be loaded into register A. Figure 4.5 displays the timing diagram for the PIO signals during this operation. The sending device, P(i-1), examines the ARDY(i) line, before it sends any valid data to P(i). If found high, it resets the ASTB(i) line. This transfers the data present on the data bus to the input register of A(i). At the end of the transfer the ASTB(i) is set by P(i-1). This resets the ARDY(i) disabling the port for further data transfers. When an IN instruction is executed by the P(i), at the falling edge during T1 of the succeeding instruction, the

ARDY(i) is set by the PIO enabling the port. If the ARDY(i) line is found high when an IN instruction is in progress, is kept low by the PIO until the falling edge during T1 of next instruction.

4.3. The HOLD controller

4.3.1. General operation

From the description of the handshake lines of the PIO we can deduce two things: after the receiver has read the previous data, its ARDY is high and when the sender has sent a data to the receiver, the ARDY of the receiver is low. If the HOLD controller monitored these two lines at the time of transfers, it could easily regulate the message traffic. ARDY(i+1) will be low when P(i+1) is not ready to receive data. If at this time, P(i) tries to send another data to P(i+1) the HOLD controller will place a low on the WAIT line of P(i). This would prevent P(i) from completing the OUT instruction and keep it in a wait state until ARDY(i+1) is set. In the same manner, ARDY(i) will be high if P(i-1) has not sent any data to P(i). If P(i) attempts to read its input port register at this time, it will be forced into a wait state by the HOLD controller.

The HOLD controller should be able to recognize the attempts by either processors to send to or receive from the other in order to generate the necessary

signals. This is made possible by including some decoding logic that will produce a signal when the corresponding port is addressed for data transfer. An OUT signal is generated if port C is accessed during an OUT instruction. Similarly, an IN signal is produced if port A is addressed during an IN instruction.

$$\begin{aligned} OUT' &= WR \vee IORQ \vee CE2 \vee (C/D) \vee (A/B) \\ IN' &= RD \vee IORQ \vee CE1 \vee (C/D) \vee (A/B) \end{aligned} \qquad (4.1)$$

We observe that OUT' or IN' will be low when an OUT or IN instruction is executed addressing the particular port involved. Then the WAIT signal for P(i) on send is generated as follows (recall that the WAIT is an active-low signal):

$$WAIT(i) = OUT' \vee ARDY(i+1)$$

In the same manner, the WAIT signal for P(i) on receive is generated as follows:

$$WAIT(i) = IN' \vee ARDY(i)'$$

Combining these two equations, we get,

$$WAIT(i) = (OUT' \vee ARDY(i+1)) \wedge (IN' \vee ARDY(i)') \qquad 4.2$$

Due to some complications encountered during implementation, equation 4.2 cannot be used as it is (this will be described later). Two signals, SEND(i) and RCVE(i), are introduced which at any time define the state of the

data transfers for P(i). A low on SEND(i) means that P(i+1) has not yet read the previous value and thus P(i) cannot initiate another send. Similarly, a low on RCVE(i) refers to the fact that there is nothing to receive and hence P(i) must wait. Thus equation 4.2 stands modified as,

$$WAIT(i) = (OUT' \vee SEND(i)) \wedge (IN' \vee RCVE(i)) \quad 4.3$$

P(i), after it reads port A(i), resets its RCVE(i) to indicate that the port is empty. It also sets the SEND(i-1) in P(i-1) so the later can execute another send instruction. In the same fashion, P(i), after it has sent a data to P(i+1), resets SEND(i). It also sets the RCVE(i+1) in P(i+1) indicating that A(i+1) is full so that the later can do a receive. Figure 4.6 shows the above protocol in an algorithmic fashion.

In the implementation, an equivalent form of equation 4.3 is employed so that the available components could be used for fabrication.

$$WAIT(i) = [\{(OUT \wedge SEND(i)')' \wedge (IN \wedge RCVE(i)')'\}']' \quad 4.4$$

Apart from generating the WAIT signal, the HOLD controller is also responsible for transferring the data sent by the sender to the receiver. This is necessary because the HOLD controller bases its decision to stop P(i) from completing its receive instruction on the signal level on

RCVE(i). RCVE(i) is reset by P(i-1) at the end of the send operation as we explained earlier. To accomplish the transfer of data from sender to receiver, ASTB of the receiver is generated by the HOLD controller of the sender. This loads the data from the inter-port data bus on to port A of receiver. That is, HOLD controller of P(i) generates ASTB(i+1) to transfer the data sent by P(i) to A(i+1).

Thus, the HOLD controller of P(i) generates six signals: OUT', IN', SEND(i)', RCVE(i)', ASTB(i+1) and WAIT(i). Figure 4.7a displays the timing relation between these signals during an OUT instruction. Here it is assumed that there are no exceptional conditions. The CRDY(i) is monitored by the HOLD controller of P(i). The raising edge on CRDY(i) signifies the completion of the output instruction and the presence of valid data in the inter-port data bus. This is used to generate the ASTB(i+1), which transfers the data to A(i+1). The raising edge on the ASTB(i+1) sets RCVE(i) indicating that P(i+1) has a data to receive. SEND(i) is reset by the raising edge on OUT'. Note that WAIT(i) remains inactive throughout the data transfer.

Figure 4.7b shows similar timing relations for the case of an IN instruction. Here also the absence of exceptional conditions is assumed. The raising edge on IN' indicates the completion of the input instruction. This

is used to reset RCVE(i) to disable P(i) from executing another receive. It is also used to set SEND(i-1) so that P(i-1) is enabled to do another send. The WAIT(i) is not affected during the transfer.

Figures 4.8(a,b) illustrate the cases where there is an exceptional condition. In figure 4.8a, the WAIT(i) is reset the moment OUT' is generated because SEND(i) is low. When SEND(i) is set by P(i+1) (figure 4.7b), WAIT(i) is set by the HOLD controller releasing P(i) from the wait state. The send operation is completed and the data is strobed into A(i+1). Similarly, as shown in figure 4.8b, the WAIT(i) is properly generated when P(i) initiates a receive when RCVE(i) is low. P(i) exits the wait state when RCVE(i) is set by P(i-1) after it has sent a data (figure 4.7a). Figure 4.9 shows the final circuitry as it was built and tested.

From the above discussions, we observe that the only "external" signal HOLD controller uses, is the CRDY(i) produced by PIO of P(i). The HOLD controller uses this signal to generate ASTB(i+1). However, a close examination of figure 4.7a will reveal that the generation of ASTB(i+1) could have been triggered by the raising edge on OUT'. This means that the HOLD controller could have actually been built independent of the PIO unit and attached to the CPU as a separate peripheral device. The ASTB signals, though are part of PIO, would be still necessary to

serve as latching signals in a design that does not use the PIO.

The next section discusses some of the problems encountered during the implementation of the HOLD controller. Beginning with equation 4.2, it shows how equation 4.3 is derived. Though the problems were not very difficult to solve, their solutions did lead to a design that could function independent of the PIO as we explained above.

4.3.2. Problems and solutions

Let us assume for the time being that we are trying to implement equation 4.2 as it is. Suppose that P(i) has just completed a receive operation. Referring to figure 4.5, we see that ARDY(i) is set by the PIO at the end of the execution of the corresponding IN instruction. Let us also assume that P(i) is initiating another receive operation. IN' will be low and since ARDY(i) is high, by equation 4.2, WAIT(i) will be reset by the HOLD controller. This forces P(i) into a WAIT state. P(i) should continue to be in the wait state until ARDY(i) is reset by P(i-1) after it has sent the next data. However, P(i) does not remain in the wait state as expected. As shown in figure 4.10, P(i) is prematurely released from the wait state because, the PIO resets the ARDY(i) line - ARDY(i) becomes low - during the execution of an IN instruction as

explained earlier (see figure 4.5 and section 4.2.2). This results in P(i) reading the previous data as many times as IN is executed by P(i) before the next valid data arrives from P(i-1). The sender does not suffer from a similar set back because it makes use of ARDY line of the receiver rather than the CRDY of its own PIO.

To circumvent this situation, the RCVE(i) signal is introduced in place of ARDY(i). RCVE(i) is made to reflect the state of the input port at any time: a high indicating the presence of valid data and a low representing an empty port. RCVE(i) is set by P(i-1) after it strobes in the data using ASTB(i). RCVE is reset by P(i) after it completes the receive operation. Thus RCVE(i) removes the circuit's dependency on ARDY(i) and prevents P(i) from exiting the wait state inappropriately. However, this arrangement introduces another problem: deadlock.

Suppose that P(i) begins the execution of an IN instruction when RCVE(i) is low. The HOLD controller will promptly force P(i) to enter a wait state. P(i) will be maintained in wait state until RCVE(i) is set by P(i-1). But, the ARDY(i) line is still reset by the PIO and kept low until the end of execution of the IN instruction. Now, if P(i-1) tries to do a send, it will be blocked from completing it because A(i) is low (figure 4.11). The result is deadlock.

The onset of deadlock can be prevented if the sender's direct dependency on the receiver's ARDY line is removed. The SEND(i) signal is introduced to replace ARDY(i+1) in equation 4.2. A high on SEND(i) means that A(i+1) is available for another data. A low indicates the opposite. SEND(i) is set by P(i+1), after it has received the previous data. It is reset by P(i) after a successful completion of a send operation.

RCVE and SEND signals could have actually been integrated into one signal for the HOLD controller. One signal would have sufficed in this case because the send and receive operations are not permitted to occur at the same time. However, in the next design, as we will see, they can indeed occur at the same time and hence a single signal would have introduced a "critical section". Just to make the generalization easier, we introduced both RCVE and SEND in the HOLD controller itself.

A power-on reset circuit is included that will initialize the SEND and RCVE signals to high and low respectively. At power-on time, this allows the uninterrupted execution of the first send operation. It also ensures that first receive instruction is blocked if no data has been sent by the sender yet.

There is one problem the current design does not address: the Z-80 available in the laboratory uses

dynamic RAM which requires a refresh signal every 2 ms. The manufacturer recommands a time gap of 1 ms between successive refresh cycles. This means, the CPU cannot be held in a wait state for more than 1 ms since the refresh cycles are generated by the CPU itself during the course of instruction executions. However, it should be mentioned that a period of 1 ms represents, on the average, the execution time of about 100 Z80 instructions. If the programs consist of more than 100 instructions, then it is recommended that the dynamic RAM be replaced by a static RAM to avoid loss of data in the presence of extended wait states. The loss of data can also be prevented by including a refresh circuitry.

4.3.3. Technical details

Figure 4.9 is the circuit diagram of HOLD controller. The address decoding is done by 7485 comparator and 74LS01 NAND gates to produce CE1 or CE2, A/B and C/D. 74LS05 hex inverter, inverts these signals and IORQ, WR signals. These are input to 7421 AND gates to produce OUT and IN signals. 74121 monoshot generates ASTB(i+1). Note that the ASTB is generated as an active high signal. The inverters on the PIO unit converts this signal into an active low signal.

The positive pulse output from the 74121 is 160 ns long which is 10 ns more than required for strobing the

data bus onto input register of port A(i+1). The SEND and RCVE signals are derived from two flipflops (74LS73A). The falling edge ASTB(i) is used to set the RCVE flipflop. The falling edge on IN triggers a monoshot that generates a clear pulse for the RCVE flipflop. Similarly, the falling edge on IN from the receiver, P(i+1), is used to set the SEND flipflop. The falling edge on OUT, triggers a monoshot that clears the SEND flipflop. (see figures 4.7(a,b) and 4.8(a,b)).

74123 is used to generate the clear pulses for the SEND and RCVE flipflops. These clear pulses are 100 ns long. This means that these two signals will be low for a period of 100 ns from the moment they are cleared. The next processor should not be allowed to set them during this period because, when the clear pulse is active, the flipflops ignore all the inputs. A close examination of the timing diagrams of 4.7(a,b) and 4.8(a,b) will reveal that such a situation will never arise.

For a more complete implementation, 74LS76A should have been used in place of 74LS73A because the former has a preset facility whereas the later doesn't. The preset and clear features are required to implement the power-on reset circuitry. The reset signal from the CPU can be used to set the SEND flipflop and reset RCVE flipflop at the power-on time and at every manual reset of the entire system. Since 74LS76A and other flipflops with preset and

clear features were not available at the time of construction of the HOLD controller, the power-on reset facility could not be implemented.

4.3.4. Testing of the HOLD controller

The configuration in which the HOLD controller was tested, consists of two Z80 microprocessors with their PIO units interconnected. C(1) is connected to A(2) and C(2) is connected to A(1). Two copies of the HOLD controller were built; one was attached to each processor. Two methods of testing were used. In the first, called the static testing, the generation of appropriate signals were monitored for operator assisted communication. The program in each CPU, waits in a loop until issued a command to send or receive, from the terminal key board. Depending upon the prevailing conditions, the processor is prevented from completing the instructions when the buffer is full or empty. In the second, referred to as dynamic testing, the programs are let run freely without any operator intervention. One of the processors is deliberately made to be slower than the other by including extra instructions. The WAIT line of the faster processor is examined for the presence of wait signal during the execution of the I/O instruction and the length of the wait state is observed to be equal to the execution time of the extra instructions on the other processor. Figure 4.12 shows the

program used for the dynamic testing.

The sender, after setting up the PIO, begins to send continuously at an interval of approximately 10 us. The receiver receives a character approximately every 164 us. The large time delay in the receives is obtained by including extra instructions in the receiver's program. Fifteen instructions (EX (SP),IX) each requiring 10.35 us were included to introduce a delay of 155.25 us. Thus the delay between successive receive instructions is about 164 us (including the time required to do a branch). This makes the HOLD controller associated with the sender to force the sender into a wait state for approximately 154 us before every send excepting the first one. In the same fashion, the sender was made slower than the receiver and similar results were obtained. Figure 4.12 shows the program only for the case in which the receiver is slower than the sender.

The next section presents a design where a buffer of size more than one is utilized. The Am2812, a first-in, first-out (FIFO) buffer is used for this purpose.

4.4. The FIFO circuit

The HOLD controller described previously represents data transfers between adjacent processors using a buffer of size one where the buffer was part of the PIO unit. In this section, a mechanism using external buffers of size

more than one is presented. The PIO is still used to interface to the CPU. That is, the CPU still outputs to port C and inputs from port A. However, the FIFO buffer, and the associated circuitry handle the routing of the data from the sender to the receiver. The FIFO buffer used, Am2812, can contain up to 32 bytes of data.

The next section briefly describes the Am2812 followed by a discussion of the FIFO circuit. The last section describes the differences between the FIFO circuit and a method of using the private memory associated with P(i) to form an internal buffer. In the discussions to follow, a slot is used to represent one internal unit of the FIFO buffer that can hold one byte of data. For instance, Am2812 contains 32 slots.

4.4.1. The Am2812

The Am2812 can hold up to 32 bytes of data. Once the first slot is filled up, the data ripples through the buffer towards the output side and occupies the vacant slot next to a full slot. If the buffer is empty, then the data falls through the buffer to the output pins. It takes 10 us for the data to ripple through the buffer from the input side to the output side when the buffer is empty. This is referred to as the ripple through time.

There are two signals associated with either end of the buffer. IR (Input Ready) is an active-low signal which

indicates if the first slot is empty. When the buffer is full, IR will be high. PL (Parallel Load) signal loads the data present at the input pins onto the first slot. OR (Output Ready) indicates that the output pins contain a valid data. When the buffer is empty, OR remains low. PD (Parallel Dump) signal dumps the contents of the last slot onto the output pins. This action is followed by the shifting of the data inside the buffer towards the output side by one slot. PL, OR and OD are active-high signals. During the transfer of data either from the input pins to the first slot or from the last slot to the output pins, the corresponding ready (IR, OR) signals remain low.

Out of these four signals, the FIFO circuit must be able to generate PL for loading the data sent by te sender into the buffer and PD for reading the last slot to provide data for the receiver. Figure 4.13 shows the timing relations for the Am2812. PL is a positive pulse of width at least 100 ns. It is to be generated after the data on the input pins have stabilized. Similarly, PD is a positive pulse of width 100 ns. The PD should be present after the input port of the receiver becomes available for another data. The FIFO generates the PD after the receiver has read its input port so that the data at the last slot in the buffer can be routed to that input port. From the start of PD, it takes about 900 ns for the data to move from the last slot to the output pins. Similarly, after

PL has been generated, it requires about 550 ns for the first slot to become empty after the loaded data has shifted towards the output side.

4.4.2. Operation of the FIFO circuit

Figure 4.14 shows a block diagram of the FIFO circuit. The operation of the FIFO circuit is very similar to that of the HOLD controller. The SEND and RCVE signals determine at any time if the initiated operation, send or receive, can be completed. A low on SEND(i) means that the first slot of the FIFO buffer is not available for loading (buffer of P(i) is full). A low on RCVE(i) indicates that there is no data present in the input port register (buffer of P(i-1) is empty). Both these conditions signify that the P(i) must wait. When P(i) begins a send operation, OUT' is generated[7]. If the SEND(i) is found low at this time, then WAIT(i) is produced. Similarly, during a receive operation, IN' is generated. WAIT(i) is produced if RCVE(i) is low. Thus equation 4.3 holds. The difference between the operation of the two circuits is the way SEND and RCVE signals are set and reset.

Figure 4.15 shows the timing relations for the operations discussed below. The falling edge on OUT' is used to trigger a monoshot that produces PL(i). This also

[7] OUT' and IN' are described by equation 4.1.

resets the SEND(i). The IR(i) is deactivated by the Am2812 during the period in which the data from C(i) is loaded into the first slot. IR(i) is re-activated by Am2812 after this data has moved away towards the output side emptying the first slot. If the loading of the first slot fills the buffer, then IR(i) will not be activated until the first slot becomes empty. The raising edge of IR(i) is used to set SEND(i) enabling P(i) to initiate another send. In the same fashion, when P(i) initiates and completes a receive, a negative IN' pulse is produced by the FIFO circuit of P(i). The raising edge on IN' signifies the end of the input operation after which the input port (A(i)) becomes available for another data. This resets RCVE(i) to disable P(i) from initiating further receive operations. This IN' signal is also monitored by the FIFO circuit of P(i-1). The PD(i-1) pulse is generated by this FIFO circuit, which dumps the contents of the last slot of its buffer on to its output pins. The OR(i-1) is inactive during this operation. Once the dumping has been completed, the raising edge on OR(i-1) is used to trigger the generation of ASTB(i) which loads the data into A(i). The raising edge on ASTB(i) which signifies the completion of the transfer, is used to set RCVE(i). The raising edge on OR(i-1) would not be present if the corresponding buffer was empty.

As we mentioned earlier, it requires about 550 ns for the data to be loaded into the FIFO buffer. This consti-

tutes an upper bound on the frequency of the output instructions that can be executed by the CPU. However, the Z80 in operation requires about 4.4 us before it can initiate another send which is very large compared to 550 ns. On the receiver side, the FIFO takes about 900 ns to move a new data to its output pins. It requires another 150 ns to strobe this data into the input port register of the receiver. Consequently, there must be a delay of at least 1.05 us between successive input instructions. Again, the large time delay does not concern us because the CPU in use is slow. Figure 4.16 shows the final FIFO circuit. Due to time constraints, this could not be built and tested.

As in the case of HOLD controller, the FIFO circuit could have been built without the use of the PIO unit. Referring to figure 15, we see that the FIFO circuit does not make use of any of the handshake lines associated with the PIO except for the ASTB.

4.5. A method of using internal buffer

In this section we will describe a method of housing the buffer in the private memories of the processors instead of keeping them external as in the case of above two designs. Though this offers a flexibility of alteration of the buffer size under program control, we will see that the FIFO circuit fares better than this method in

terms of speed of operation.

In the internal buffer method, a part of the memory M(i), of P(i) is set aside as a circular buffer to store the data sent by P(i). The associated circuitry, called the DMA circuit, would access M(i) to store the data and later retrieve it when P(i+1) requires it. The storage and retrieval operations are carried out by doing a direct memory access. This has to be done in three parts: first, issue a bus request (BUSRQ) to the CPU and wait until the CPU relinquishes the bus; next, access the memory to either store or retrieve a data; and lastly, update the pointers so that buffer full or empty conditions can be detected. Two pointers have to be maintained, one to keep track of the next available storage location (HPTR) and the second to point to the next available data for retrieval (TPTR). When HPTR is found equal to TPTR on a store operation initiated by P(i), it signifies a buffer full condition. P(i) has to be halted at this instant from completing the send operation. Similarly, on a receive command from P(i+1), if TPTR is null, then the buffer is empty and hence P(i+1) has to be halted. The halting cannot be done using the WAIT line, because, when the WAIT line is active, the CPU maintains its control over the bus. Thus the DMA circuit would be unable to enter the processor's memory to store or retrieve a data which will be necessary to remove the exceptional condi-

tion. The only way the DMA can operate now is to issue a BUSRQ and wait until the processor releases the bus lines. Once they are released, the DMA circuit must assume control over them and retain it until the exceptions are removed. This will be done at the end of execution of the current instruction. However, the completion of the instruction lets the sender lose its data on a buffer full condition and allows the receiver read an invalid data on a buffer empty condition.

For the sender, the lose of data can be prevented by adding an extra latch outside the memory to hold the data until the location at HPTR becomes empty. For the receiver, the reading of the incorrect data can be avoided by forcing it to execute another IN instruction by means of interrupts or having two IN instructions in tandem throughout the program. The first instruction will give advance information to the circuit about the intentions of the CPU and the second will read a valid data. When the first option is used, an IN instruction within the interrupt service routine would accomplish the task. The choice of using interrupts is not very desirable because it increases the program execution time by 14 us[8]. The alternative of having doubled IN instructions in the pro-

[8] This includes the time to recognize the interrupt, time to branch to the ISR and the time to execute an input instruction.

gram is also equally undesirable because it increases the program execution time by 4.4 us[9] for every input value to be read.

Also during normal operations, when there are no exceptional conditions, for every output instruction from P(i), it requires 350 ns to store the data in the buffer (the time for a memory write) plus at least another 400 ns (one T cycle) which is the time required by the CPU to regain control of the busses. Similarly, for every input instruction executed by P(i+1), a total of 750 ns is required to refurbish A(i+1). To this the time period of a machine cycle (maximum 5 T cycles or 2 us) has to be added since it is the time required by the CPU to release the busses. If there are N non-local values to be sent and received, then this represents a total stretch of N × 3.5 us in the program execution time.

In the DMA circuit, on a buffer full condition, the sender has to wait for additional 1.1 us after the input port of the receiver becomes empty[10]. On a buffer empty condition, the additional time is only 150 ns which is the time required for strobing the data into the input port of the receiver assuming that a short circuit path is

[9] 4.4 us is the time to execute an IN instruction.

[10] 350 ns to retrieve a byte of data for the receiver, 350 ns to store the data that caused the buffer full condition in the buffer and 400 ns for the CPU to resume its normal operation.

available from the sender to the receiver to route the data directly to the receiver when it is waiting.

In contrast, for the FIFO circuit, no processor time is wasted in updating the buffer. The extra waiting time on buffer full condition is the width of the OUT' pulse during normal operation (approximately 1 us) and 150 ns on buffer empty condition assuming that a similar short circuit path is available. If such a path is not present then the additional waiting time is equal to the ripple through time (10 us). Table 4.1 lists the above discussion in a tabular form.

4.6. Asynchronous processors in shared I/O

The HOLD controller and the FIFO circuit we discussed above interface two semisynchronous microprocessors. In this section, we will examine how these designs should be modified so that they could be used to interface two asynchronous microprocessors. This is to give the user of shared I/O, an option to choose between these two types of microprocessors. MC68000 microprocessor is considered for this purpose. The approach here is just to mention what additional functional units are required for MC68000 in comparison to the circuitry we built for Z80. No attempt is made to give exact details of the circuit.

MC68000 is a memory mapped microprocessor. That is, its input and output instructions are regular memory read

and write instructions. The send and receive operations of the shared I/O can be implemented using these read and write instructions. The address specified in the instruction would be the address of HOLD controller or FIFO circuit[11]. There will be one I/O controller for each processor. The I/O controller of P(i) will send data to I/O controller of P(i+1) and receive from that of P(i-1).

The communication between the CPU and the I/O controller takes place in a fully interlocked manner. For reading a value from the I/O controller, the CPU uses two control lines. The data strobe line (DS)[12] when activated indicates to the I/O controller that the CPU is ready to begin the transfer. The I/O controller loads the data on to the data bus and activates the Data Transfer ACKnowledge (DTACK) line indicating that the data is available on the data bus. The CPU accesses the data and deactivates DS. This conveys to the I/O controller that the data transfer has been properly completed and it should release the DTACK line.

---

[11]For the remainder of this chapter, HOLD controller and FIFO circuit would be referred to as the I/O controller; MC68000 would be referred to as CPU.

[12]There are two data strobe lines. UDS and LDS specify the unit of data being transferred: byte (8 bits) or word (16 bits). Here we assume that only one byte transfers are present. That is, only one of these two lines would be used. DS denotes this line.

Similarly, for an output from P(i), the activation of DS signifies the start of the operation. The edge on DS can be used to latch the data present on the data bus. At the end of loading the data from data bus, the I/O controller will activate DTACK line. When DS is deactivated by the CPU, the edge on DS can be used to deactivate DTACK. At the completion of this write operation, the I/O controller routes the data to the I/O controller of the receiver. This is achieved by loading the data on the data bus between the I/O controllers and generating a latch signal (similar to ASTB of the I/O controller of Z80). The I/O controller uses the RD/WR line of CPU to identify whether the CPU is writing a data to it or trying to read from it.

For the read operation of the CPU, if DTACK is not present within about 250 ns[13] from the moment of activation of DS, then the CPU waits for two clock cycles (250 ns). If DTACK is still not present, another two wait cycles are inserted in the current machine cycle and so on. Thus, when the buffer is empty, the I/O controller can force the CPU to wait by not activating DTACK until a new data becomes available from the sender. Similarly, on a write operation, the CPU can be made to wait by keeping the DTACK inactive if the buffer is full.

[13] Assuming a 8MHz clock with a period of 125 ns.

At this point, let us examine the similarities and differences between the I/O controller we designed for Z80 and the I/O controller for MC68000.

(1) For both the microprocessors, the corresponding I/O controllers have some decoding logic to detect an I/O operation. The decoding logic is larger for MC68000, because its address bus is 23 bits wide compared to 16 bits for Z80.

(2) For the Z80, the PIO managed the data transfers to and from the STD bus. For the MC68000, this has to be done by the I/O controller[14]. The I/O controller has to monitor the DS line and use the transitions on this line in conjunction with the output from the decoding logic to load to or load from the system data bus.

(3) For Z80, the WAIT signal is generated by the I/O controller. This is a combination of Z80-CPU signals and SEND, RCVE signals. For MC68000, DTACK is generated by the I/O controller. DTACK has to be generated for all data transfers. The generation of DTACK is accomplished by resetting a DTACK flipflop when DS is activated by the CPU. When DS is deactivated, the

[14] This would not be necessary, if the equivalent of PIO, the Peripheral Interface Adapter (PIA) is used. Here we assume that such a device unavailable to us.

flipflop can be set. The resetting of DTACK has to be delayed if SEND or RCVE signals are low during write or read operations. It is not known how complex this delaying mechanism would be.

(4) For Z80, the I/O controller generates ASTB signal to route the data from the sender to the receiver. For MC68000, we will need a similar signal. This signal would be used by the I/O controller at the receiver to latch the data sent by the I/O controller at the sender.

We conclude this brief discussion, by noting that the added logic required for the I/O controller of MC68000 is,

1) logic to load from and load to the system data bus when DS is active. The direction of loading is determined by the signal on RD/WR line.
2) a circuit to control the DTACK line.
3) decoding circuitry to decode from 23 address bits.

4.7. Summary

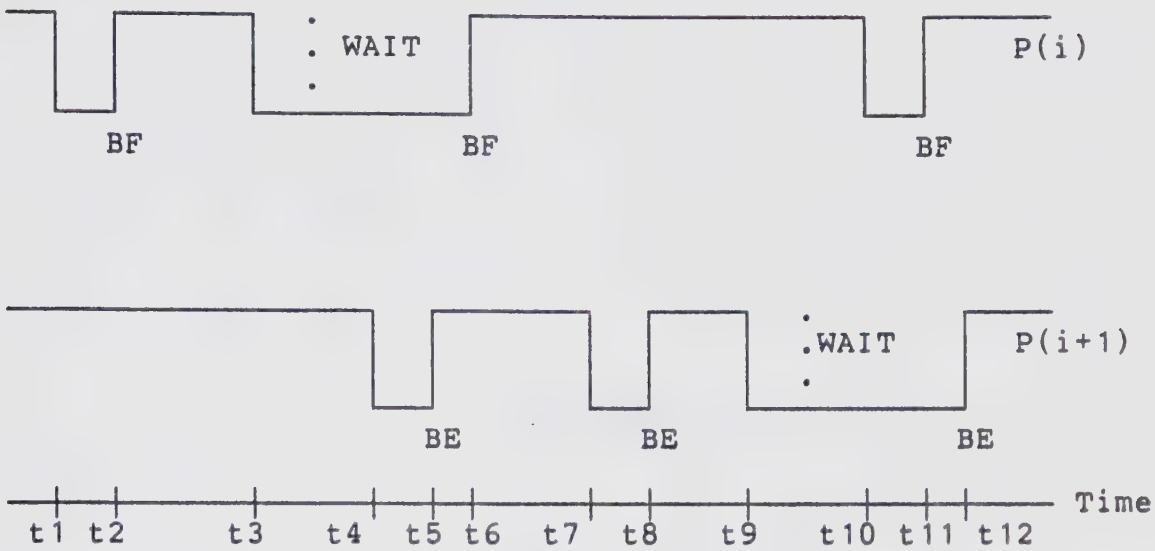
In this chapter we have discussed the design of two circuits, the HOLD controller and the FIFO circuit, to reduce the processing overhead involved in the software synchronization during data transfers between adjacent processors. As a result, Tns* is reduced increasing the system throughput. The Z80 microprocessor was used because

it had the WAIT line feature. Both circuits utilized this facility to temporarily halt the processor on buffer empty or buffer full condition. By monitoring the system control lines and the handshake lines of the PIO, the exceptional conditions are easily sensed and corrective measures taken. Though the general purpose PIO was used to interface to the CPU it is not difficult to extend these designs to function as separate I/O units interacting directly with the CPU. We also discussed, though very briefly, about the nature of the circuit required to interface MC68000 type microprocessor.

DMA versus FIFO

	DMA	FIFO
On normal operation:		
OUT:	750 ns to update the circular buffer	No processor time is required to update the FIFO memory. Requires 550 ns between successive updates.
IN:	2.75 us to load another byte of data at the input port register from the sender's memory	No processor time is required to fill same register from the head of FIFO buffer. Requires 1.05 us between successive outputs from FIFO buffer.
On abnormal conditions:		
Buffer full:	Additional waiting time is 1.1 us	Additional waiting time is approximately 1 us
Buffer empty:	Additional waiting time is 150 ns	Additional waiting time is 150 ns
Expansion	The size of the buffer can be increased or decreased under software control	A change in the wiring is required to insert or delete an Am2812 unit to increase or decrease the buffer size

Table 4.1. Comparison of DMA and FIFO techniques



t1 - P(i) initiates send 1
t2 - P(i) completes send 1
buffer becomes full (BF)
t3 - P(i) initiates send 2
must wait since buffer is full (BF)
t4 - P(i+1) initiates receive 1
t5 - P(i+1) completes receive 1
buffer becomes empty (BE)
P(i) is released from wait state
t6 - P(i) completes send 2
buffer becomes full (BF)
t7 - P(i+1) initiates receive 2
t8 - P(i+1) completes receive 2
buffer becomes empty (BE)
t9 - P(i+1) initiates receive 3
must wait since buffer is empty (BE)
t10 - P(i) initiates send 3
t11 - P(i) completes send 3 (BF)
P(i+1) is released from wait state
t12 - P(i+1) completes receive 3
buffer becomes empty (BE)

and so on

Figure 4.1. Communication protocol between P(i) and P(i+1)

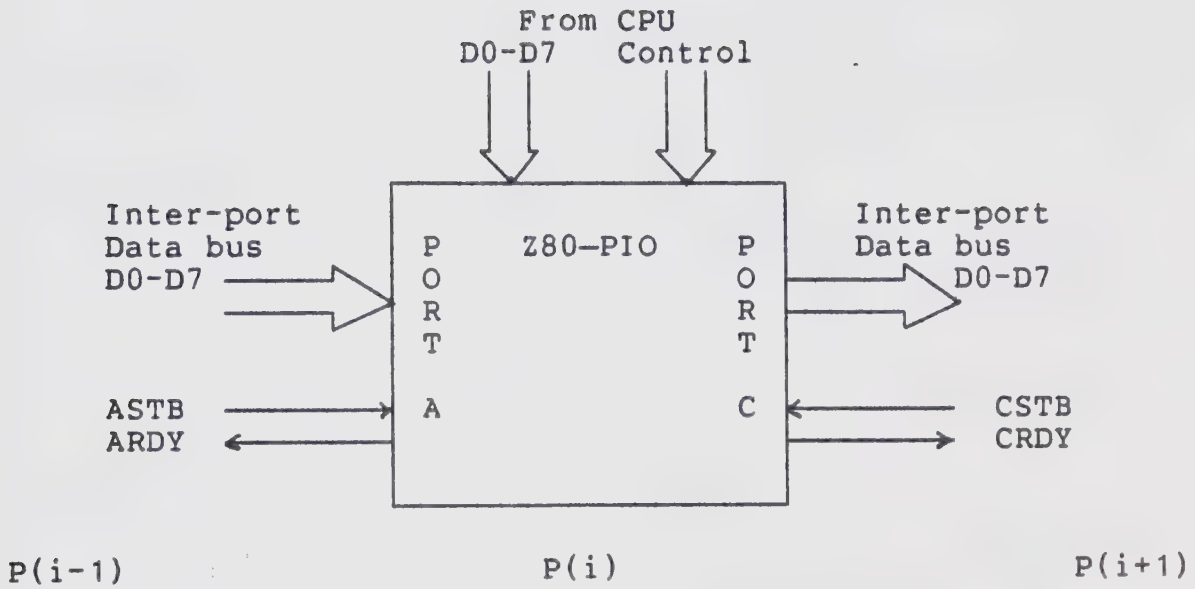
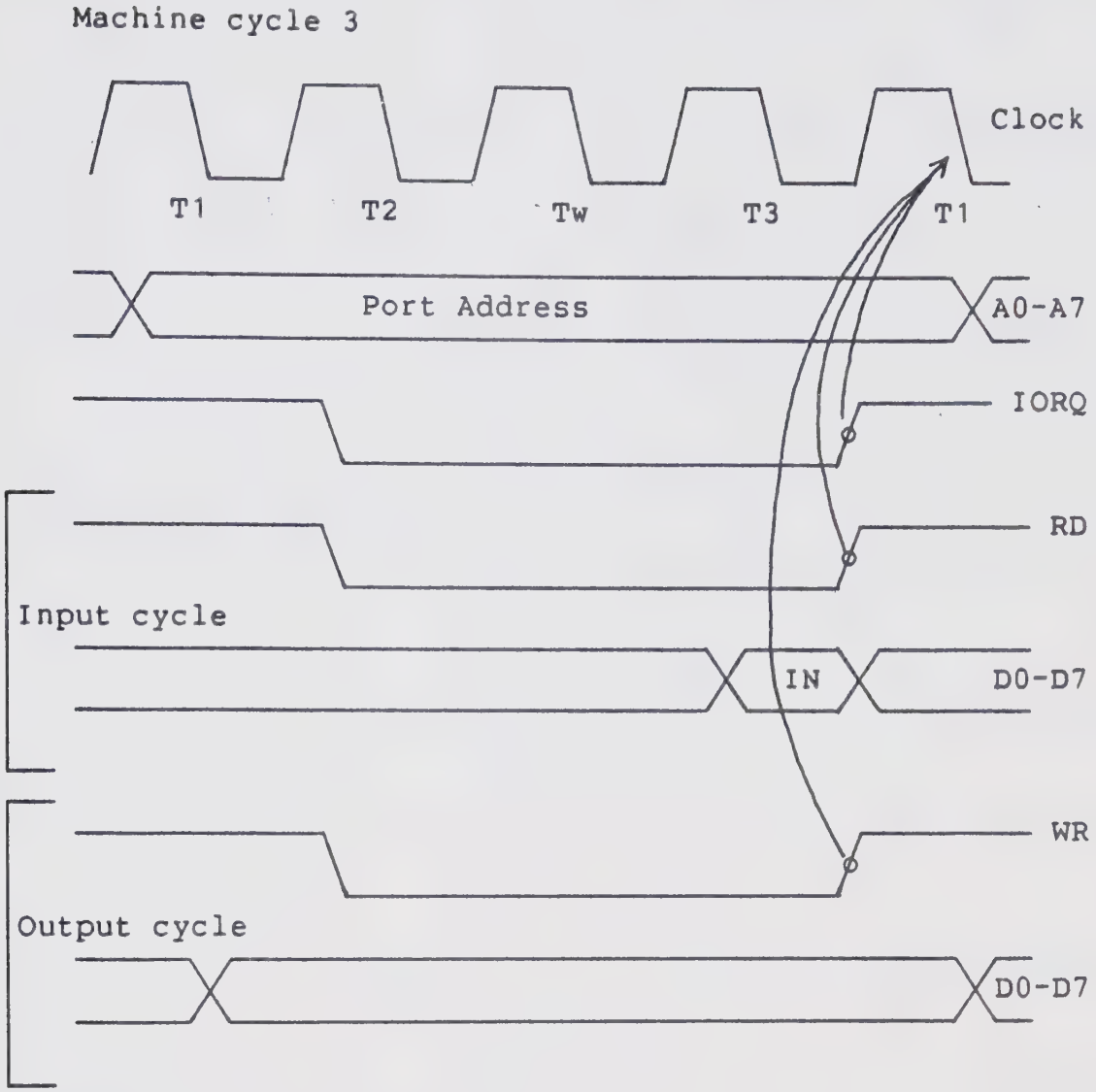


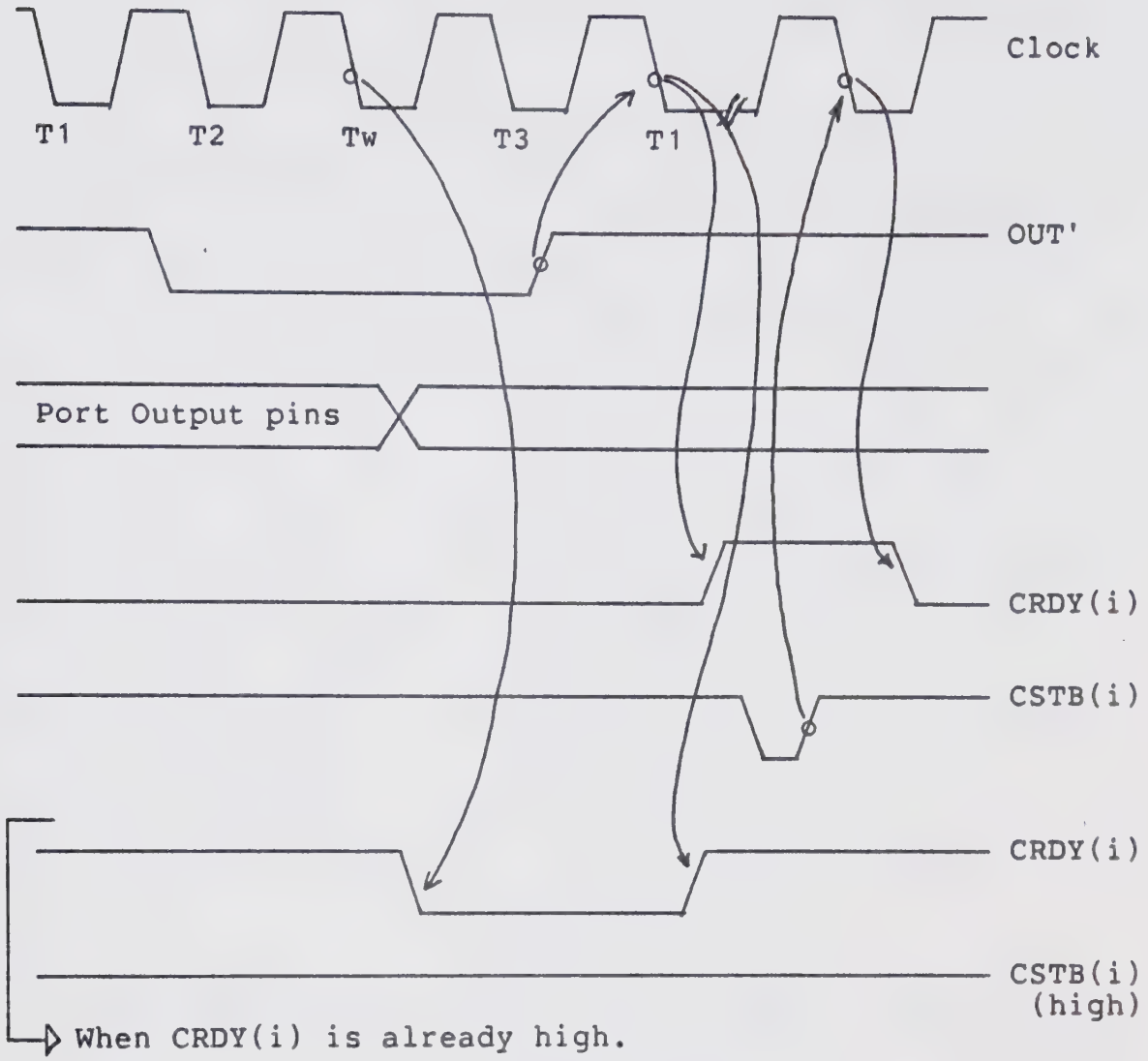
Figure 4.2. Z80-PIO as connected to P(i)



RD - Read signal WR - Write signal
IORQ I/O ReQuest
Tw - is one wait state inserted by the Z80 CPU

Figure 4.3. Input/Output cycles of the Z80 CPU without extra wait states

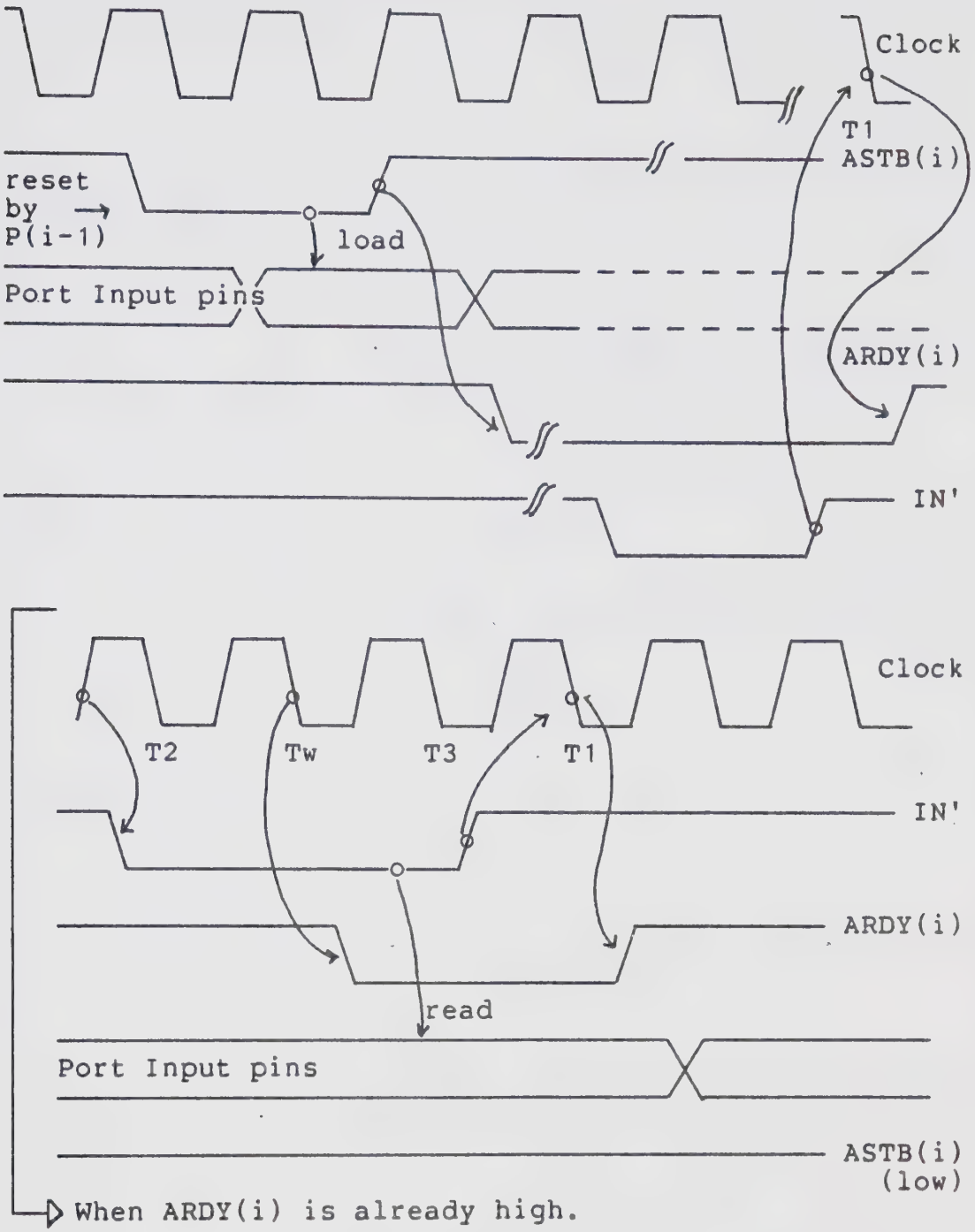
adapted from [MOST79]



OUT = IORQ' WR' CE' (C/D)' where CE and C/D are decoded from the address lines.

Figure 4.4. Timing relations on OUT instruction for the PIO

adapted from [MOST79]



IN = IORQ' RD' CE1' (C/D)' where CE1 and C/D are decoded from the address lines.

Figure 4.5. Timing relations on IN instruction for the PIO

adapted from [MOST79]

```
Sender [P(i)]:      send: OUT (nn),A

                    BEGIN send

                    load the contents of register A
                    into the output port register. nn
                    is the address of output port C(i).

                           WHILE (SEND(i) EQ  0)
                                  WAIT(i) := 0
                           END WHILE
                           WAIT(i) := 1

                    complete the transfer of data to
                    the input port register of P(i+1).

                           RCVE(i+1) := 1
                           SEND(i)   := 0

                    END send

Receiver [P(i)]:    receive: IN  A,(nn)

                    BEGIN receive

                           WHILE (RCVE(i) EQ 0)
                                  WAIT(i) := 0
                           END WHILE
                           WAIT(i) := 1

                    complete the reading of the input
                    port register. nn is the address of
                    the input port A(i).

                            RCVE(i)   := 0
                            SEND(i-1) := 1

                    END receive
```

Figure 4.6. HOLD protocol

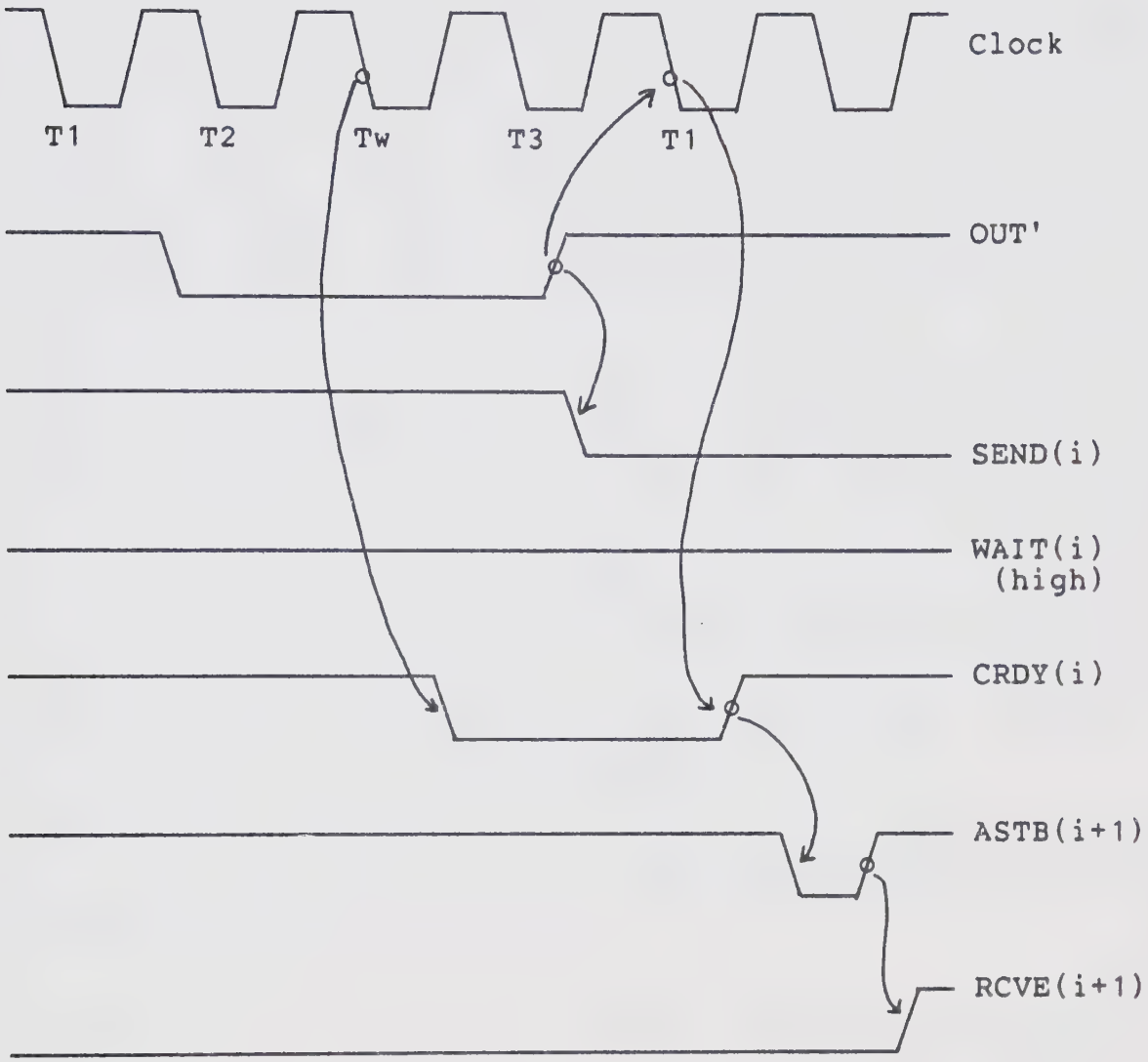


Figure 4.7a. Timing relations on OUT instruction for the HOLD controller with no buffer full condition.

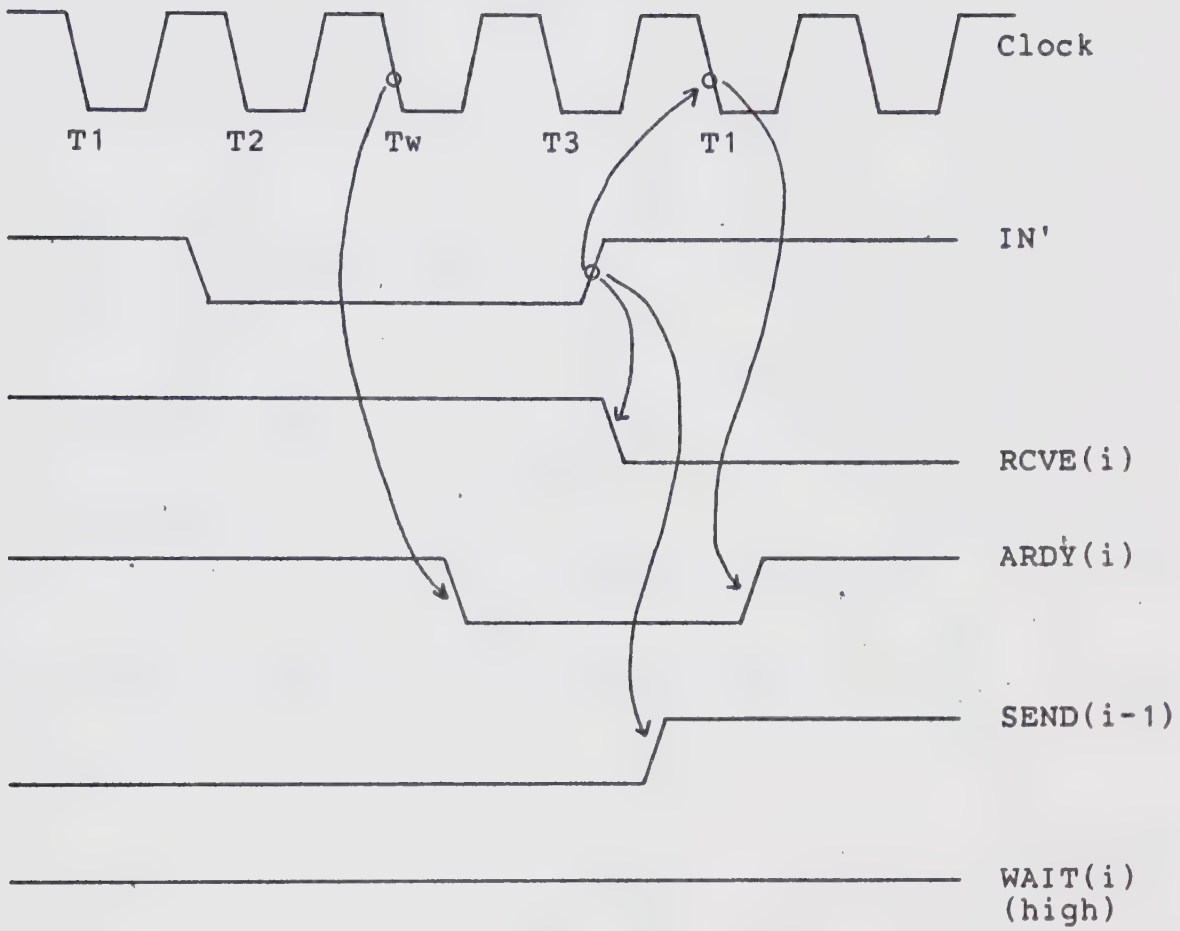


Figure 4.7b. Timing relations on IN instruction for the HOLD controller with no buffer empty condition.

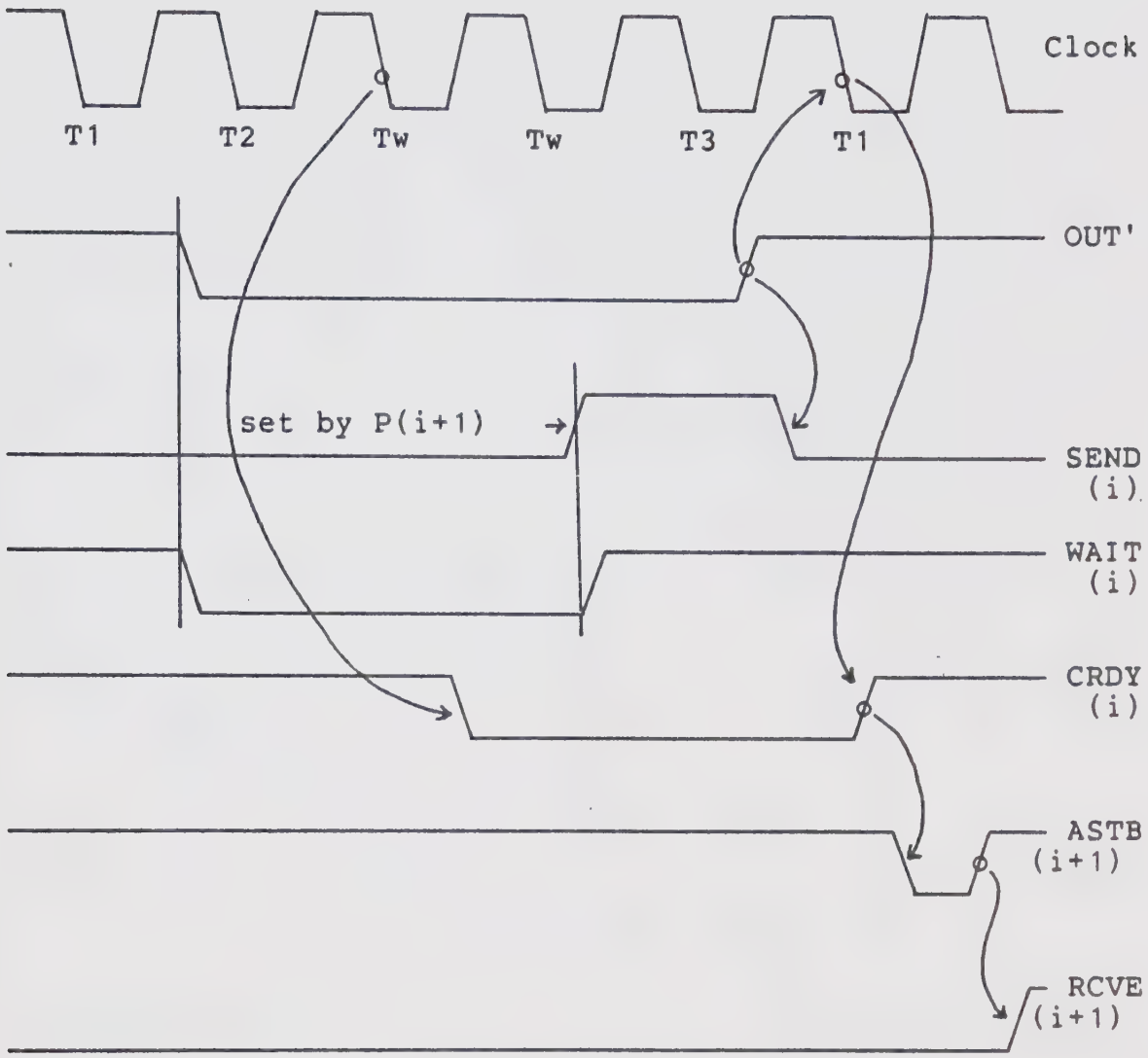


Figure 4.8a. Timing relations on OUT instruction for the HOLD controller with buffer full condition.

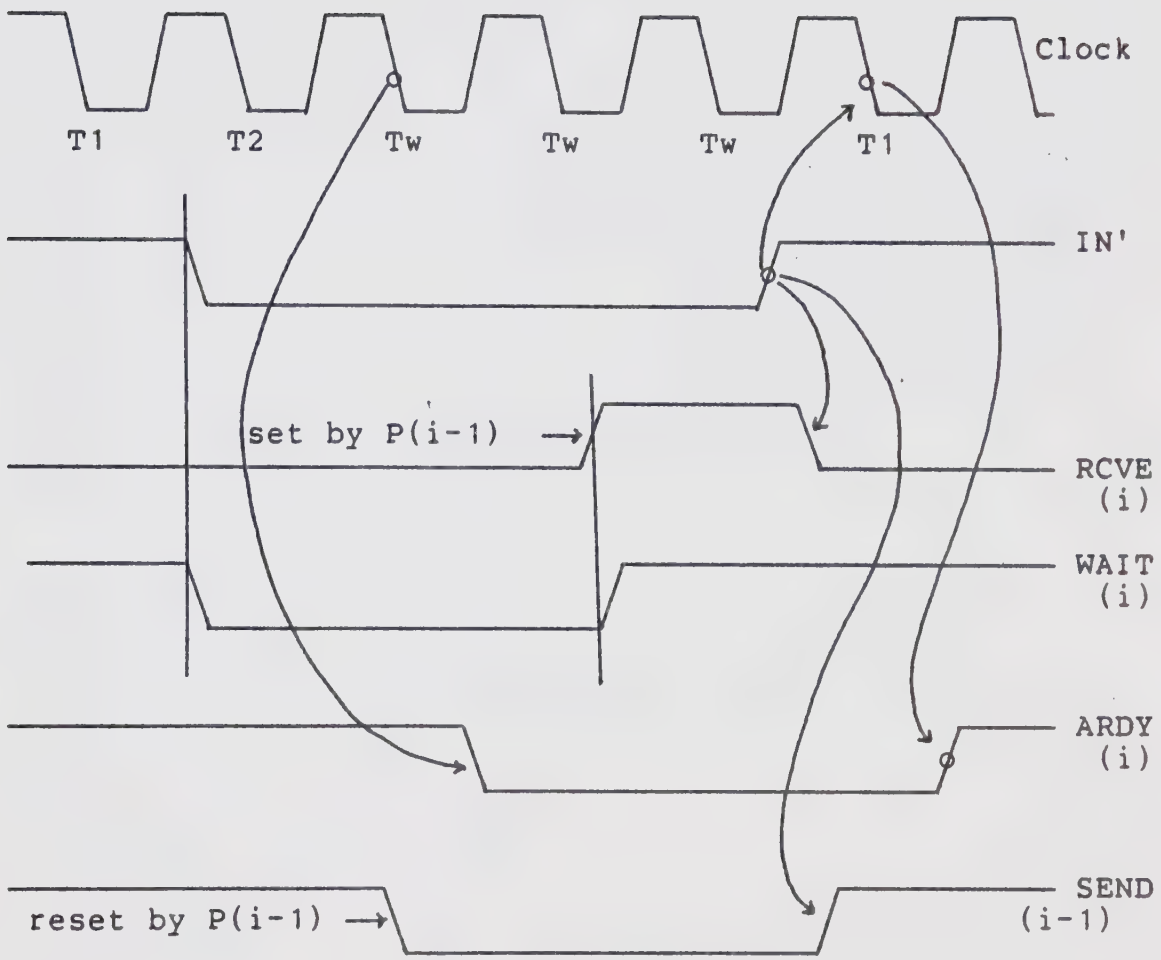
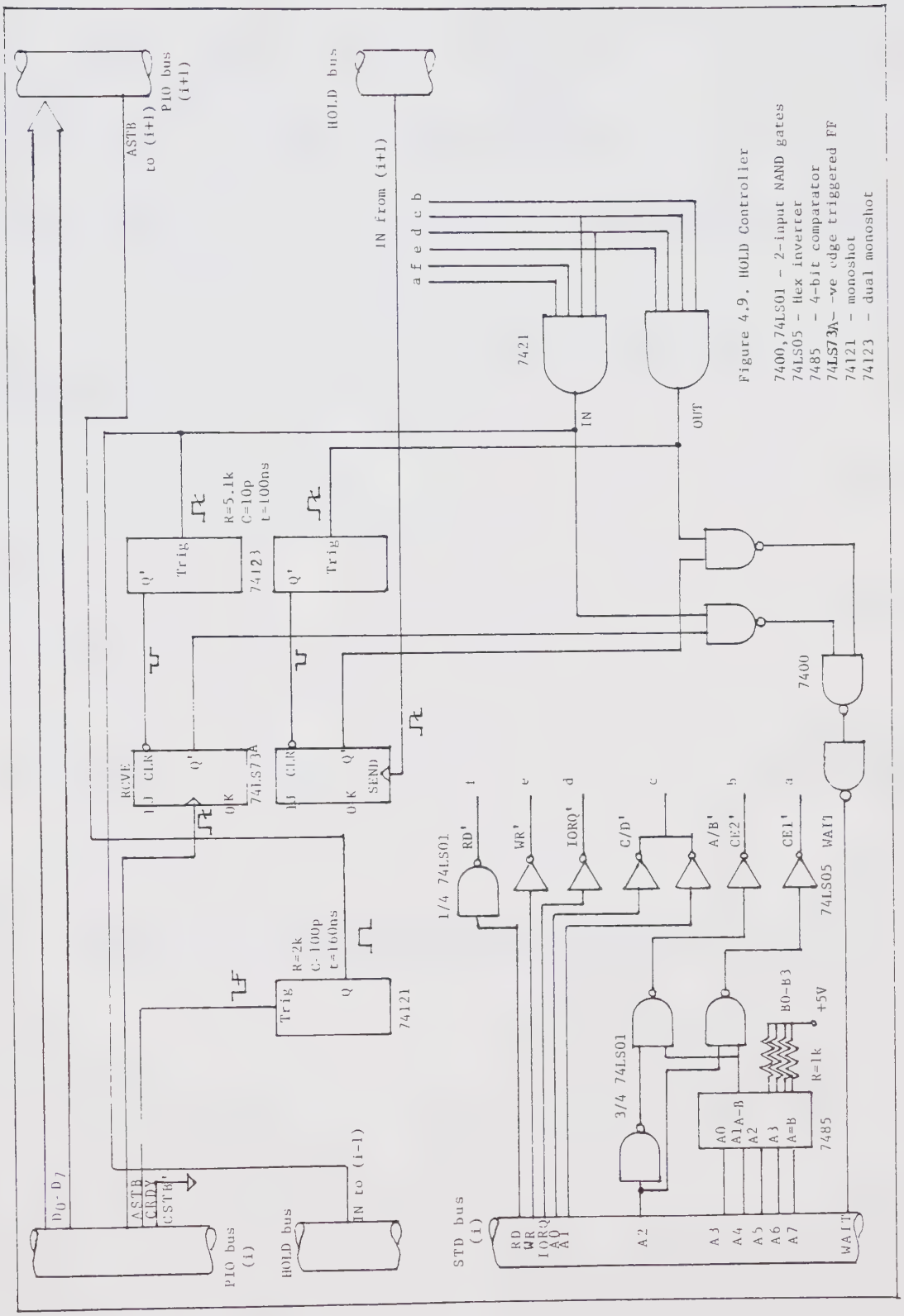


Figure 4.8b. Timing relations on IN instruction for the HOLD controller with buffer empty condition.


PIO bus (i+1)
ASTB to (i+1)
HOLD bus
IN from (i+1)
a f e d c b
7421
IN
OUT
R=5.1k C=10p t=100ns
74123
Trig
Q'
RCVE
SEND
CLR
74LS73A
7400
1/4 74LS01
RD'
WR'
IORQ'
C/D'
A/B'
CE2'
CE1'
74LS05
WAIT
3/4 74LS01
R=2k C=100p t=160ns
74121
Q
PIO bus (i)
ASTB
CRDY
ISTB'
D0-D7
HOLD bus
IN to (i-1)
STD bus (i)
RD
WR
IORQ
A0
A1
A2
A3
A4
A5
A6
A7
7485
A0
A1A-B
A2
A3
A=B
B0-B3
R=1k
+5V
Figure 4.9. HOLD Controller
7400,74LS01 - 2-input NAND gates
74LS05 - Hex inverter
7485 - 4-bit comparator
74LS73A - -ve edge triggered FF
74121 - monoshot
74123 - dual monoshot

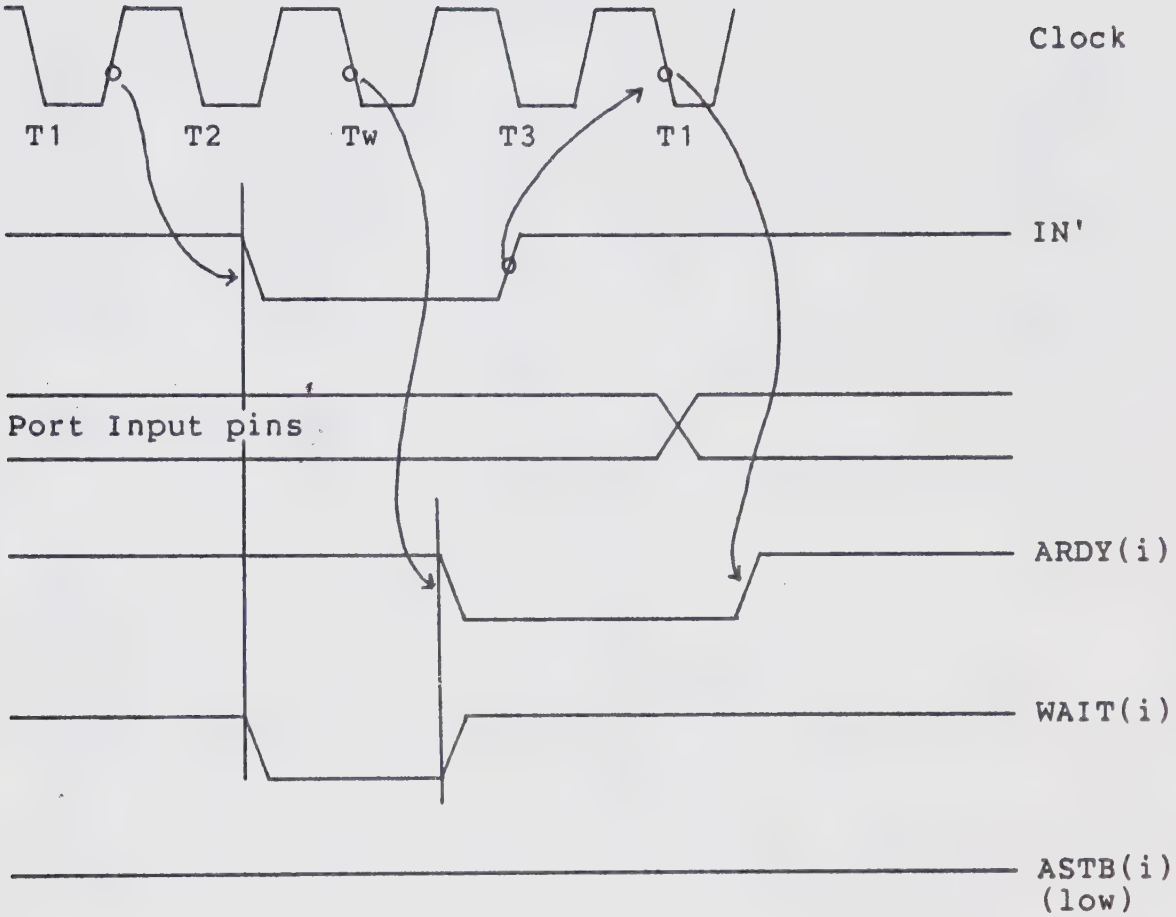


Figure 4.10. Premature release of P(i) from wait state during receive

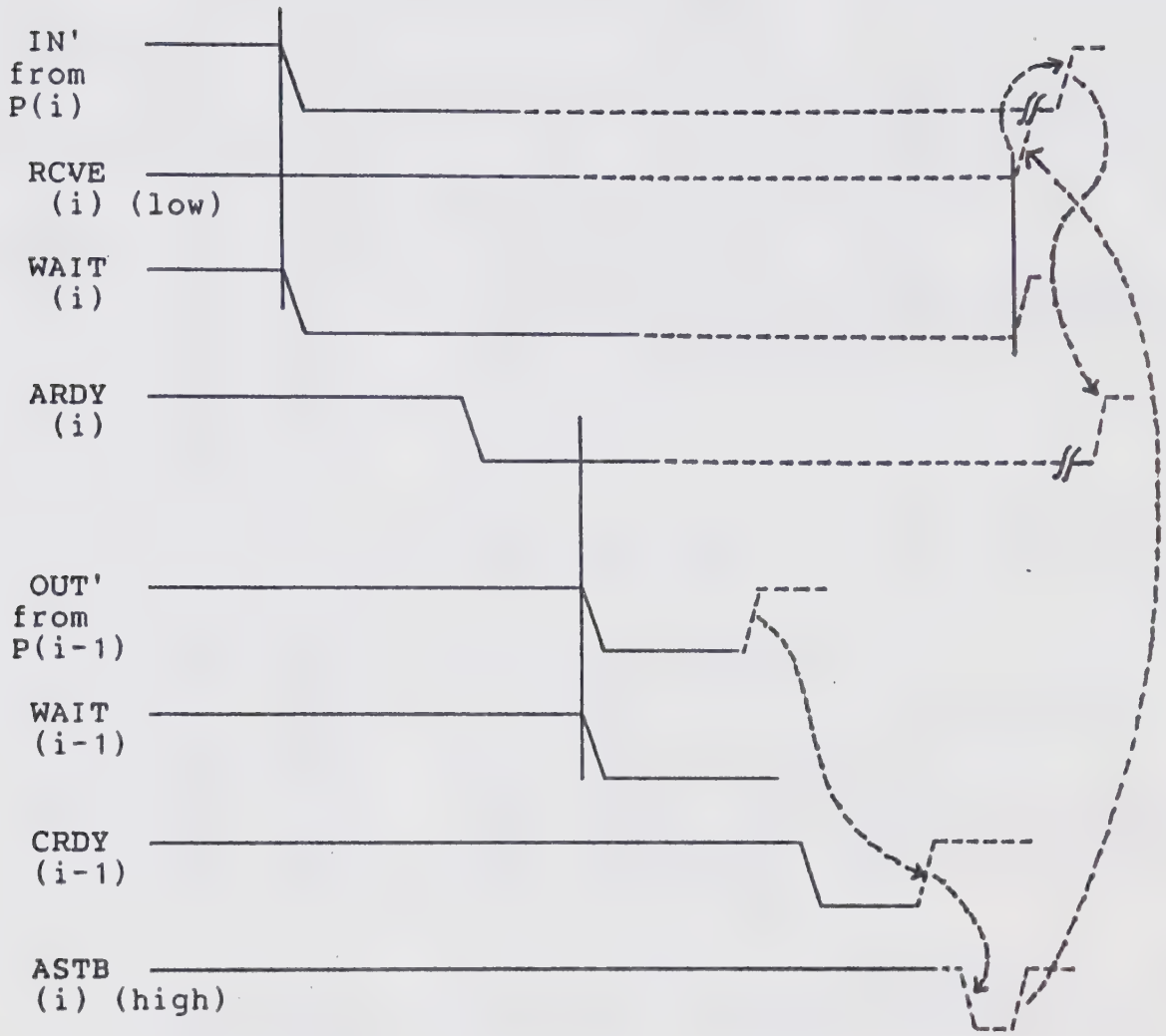
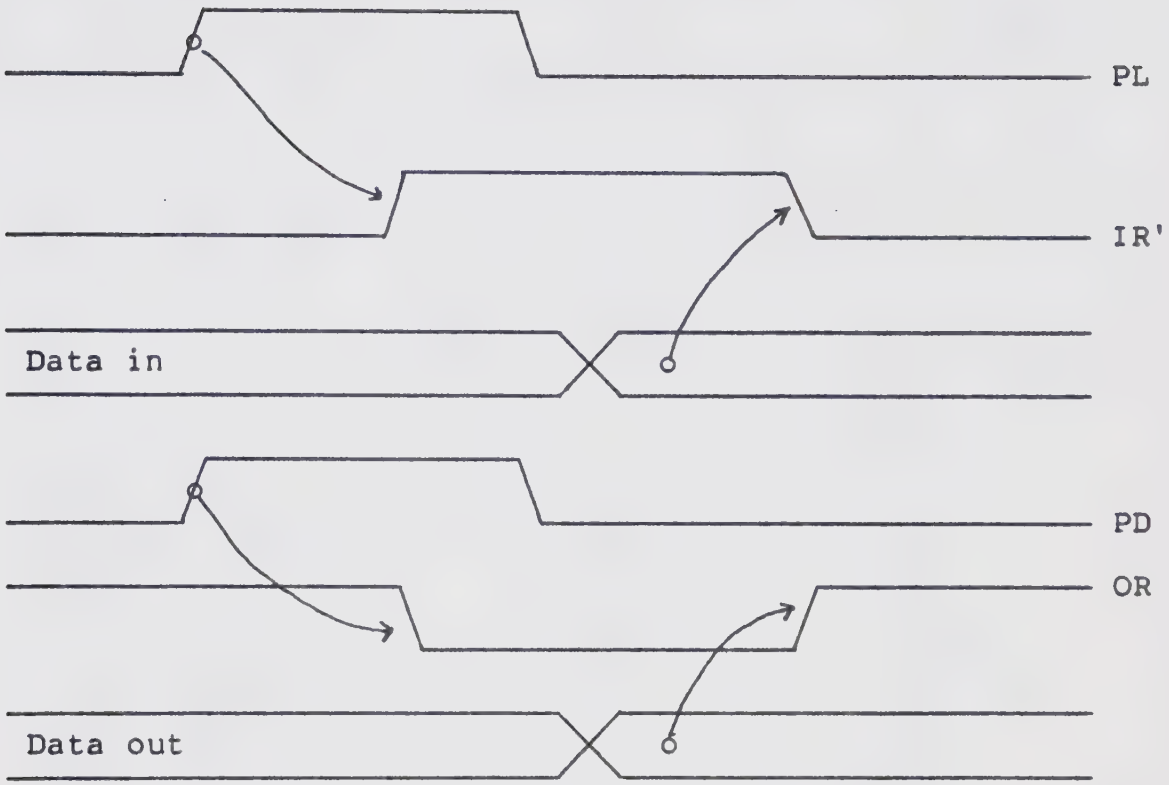


Figure 4.11. The problem of deadlock

Ignoring the WAIT(i-1), the dotted lines indicate the series of actions necessary to release P(i) from wait state.

```
;
;       Dynamic testing
;
;       This is the sender. it outputs one character every
;            10 us when within a loop
;
START   LD   A,0FH
        OUT  (0FDH),A ;C is output port
        LD   B,46H    ;70 characters to be output per loop
        LD   C,0FCH
        CALL 0E522H   ;wait for the go signal
        LD   D,21H    ;the first character is !
LOOP    OUT  (C),D    ;(delay=3.2 us)
        INC  D        ;change the character(delay=1.6 us)
        DJNZ LOOP-$   ;output 70 characters(delay=5.2 us)
        LD   D,21H    ;reset the registers
        LD   B,46H
        JR   LOOP-$   ;keep going
        HALT
        END  START
;
;       This is the receiver. it inputs one character every
;            164 us when within a loop.
;
START   LD   A,4FH
        OUT  (0F9H),A ;A is the input port
        LD   B,46H    ;70 characters to be input per loop
        LD   C,0F8H
        CALL 0E522H   ;wait for the go signal
        LD   D,21H    ;dummy instruction so that both loops
LOOP    IN   D,(C)    ;start at the same time(delay=3.2 us)
        EX   (SP),IX  ;dummy instruction introduces a delay
        EX   (SP),IX  ;of 10.35 microseconds
        .
        .             ;15 times (delay = 155.25 us)
        .
        DJNZ LOOP-$   ;delay=5.2 us(total delay=163.65 us)
        LD   D,21H
        LD   B,46H
        JR   LOOP-$
        HALT
        END  START
;
;       the time the sender spends waiting between
;            successive OUTs = 154 us.
;
```

Figure 4.12. Dynamic testing program for HOLD controller



PL - Parallel load
IR - Input ready for loading: first slot is empty
PD - Parallel dump
OR - Output ready: output lines contain valid data

Figure 4.13. Timing diagram for Am2812

adapted from [AMDI80]

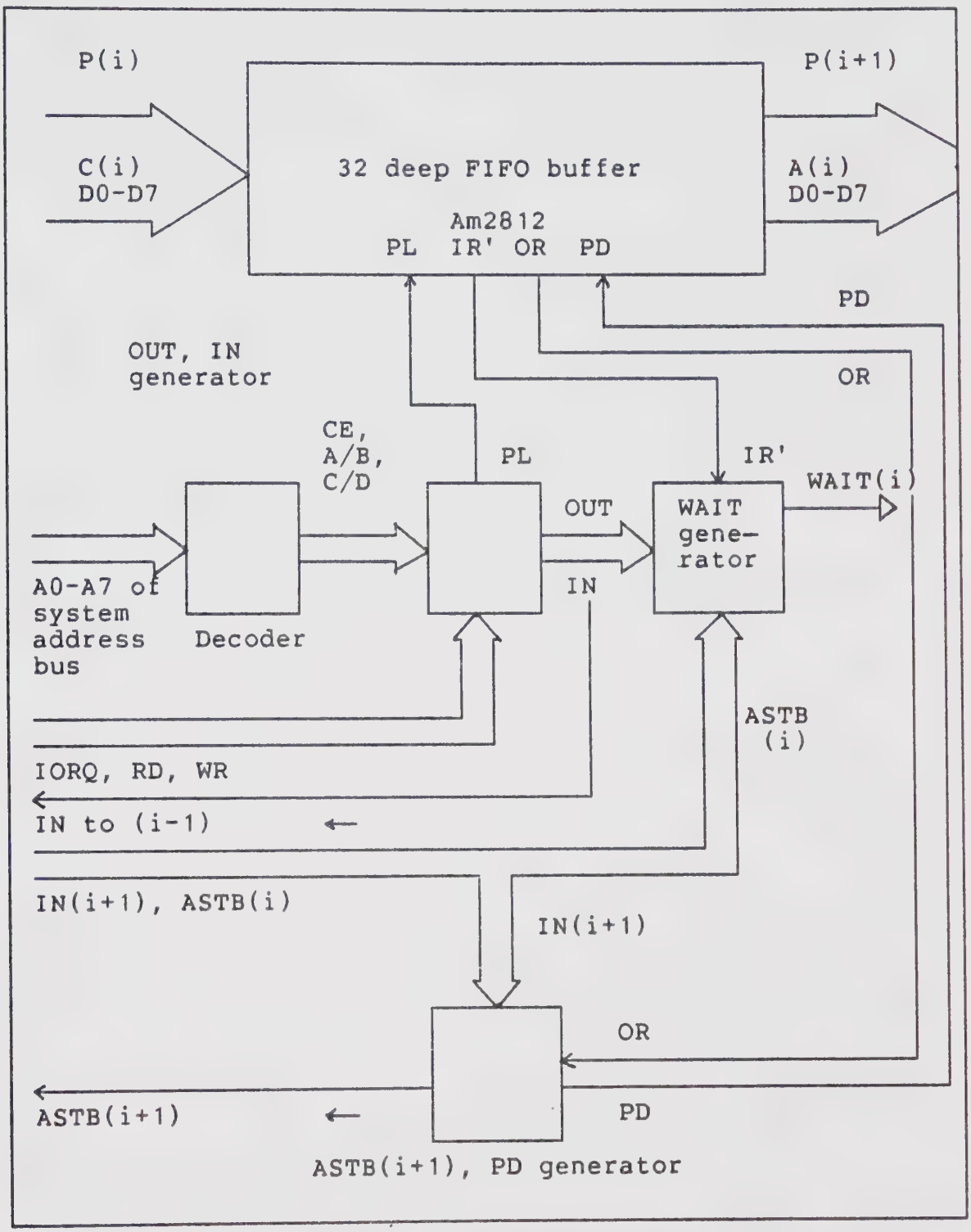


Figure 4.14. Block diagram for FIFO circuit

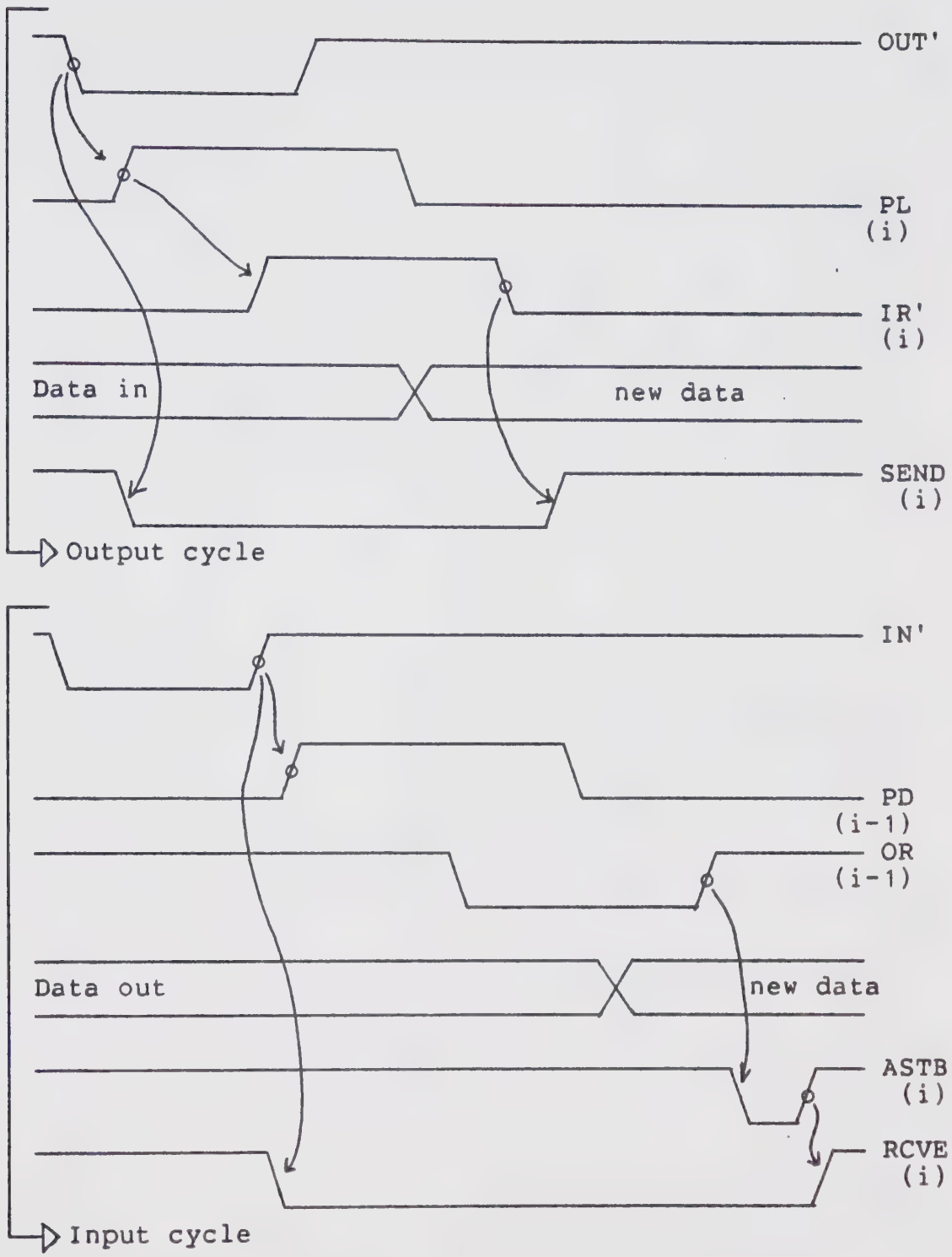
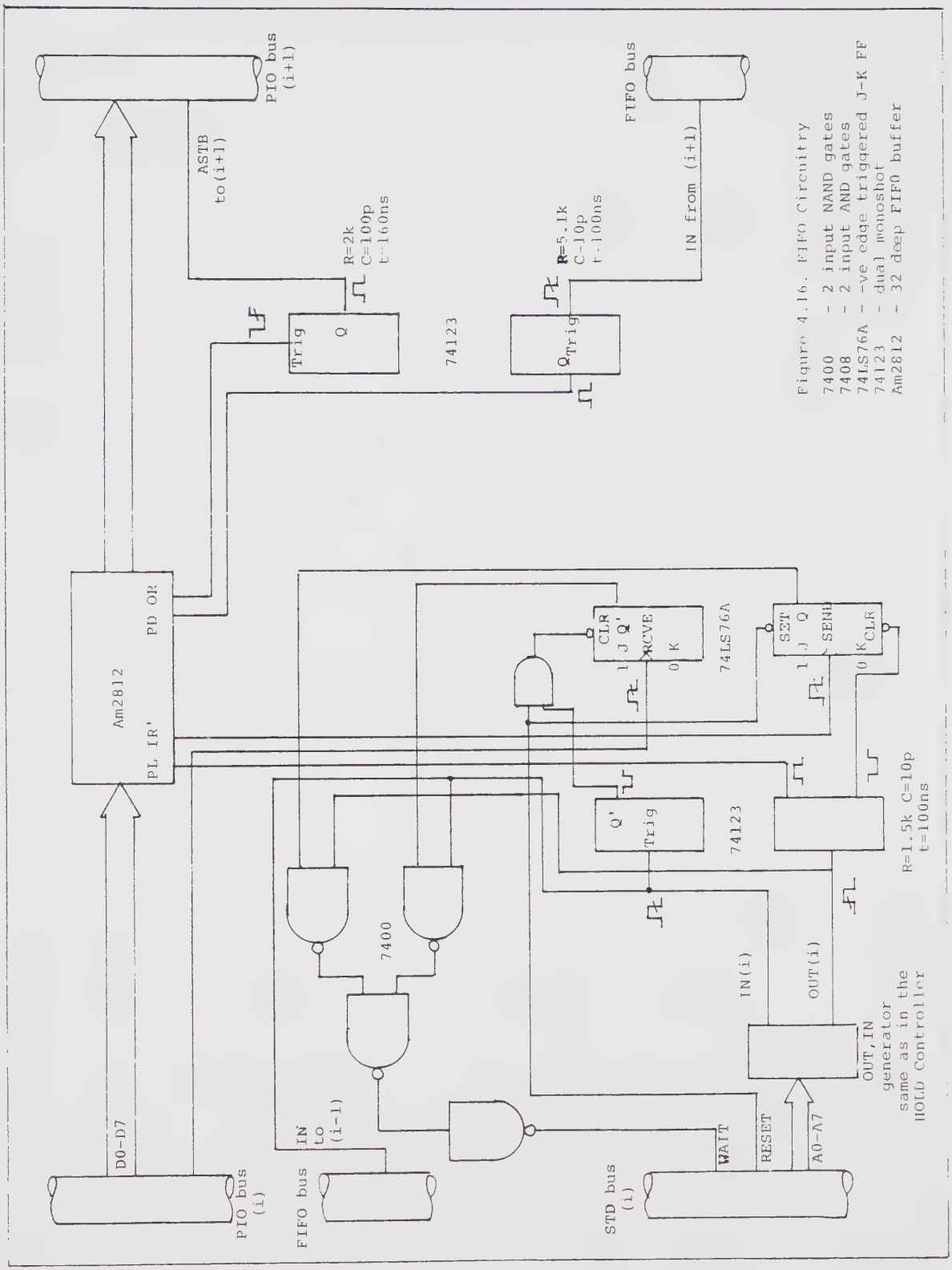


Figure 4.15. Timing diagram for FIFO circuit.


PIO bus (i+1)
ASTB to(i+1)
R=2k C=100p t=160ns
74123
Trig
Q
FIFO bus
IN from (i+1)
R=5.1k C=10p t=100ns
Q Trig
Figure 4.16. FIFO Circuitry
7400 - 2 input NAND gates
7408 - 2 input AND gates
74LS76A - -ve edge triggered J-K FF
74123 - dual monoshot
Am2812 - 32 deep FIFO buffer
Am2812
PD OR
PL IR'
D0-D7
PIO bus (i)
CLR
J Q'
RCVE
K
74LS76A
SET
SEND
CLR
Q'
Trig
74123
7400
IN to (i-1)
FIFO bus
IN(i)
OUT(i)
R=1.5k C=10p t=100ns
OUT, IN generator same as in the HOLD Controller
WAIT
RESET
A0-A7
STD bus (i)

CHAPTER 5

Conclusions

5.1. Summary

In this thesis, we have presented a description of Shared I/O organization. It consists of multiple processors with adjacent processors connected to each other. Successive repetitions of a program loop are assigned to adjacent processors such that more than one loop repetition may execute in parallel. The maximum degree of overlapping is controlled by a certain characteristic of the program loop, namely, the maximum time to execute a non-local segment, Tns*. The throughput is the same as the rate at which the input can be streamed into the system, 1/Ts, as long as satisfies two constraints:

1) $Ts \geq Tns*$

2) $Ts \leq Trs*$

The first constraint is present when Tns* is greater than zero and the second when Tns* is equal to zero.

We compared shared I/O with pipeline and pointed out that while the performance of the former is only limited by the input rate, for the later, the bottleneck of the structure forms the corresponding upper bound. We showed that both shared I/O and pipeline have similar utilization

of the available resources. The utilization is maximum when there is only one processor in the shared I/O and when there is only one stage in the pipeline. We listed additional features of shard I/O such as expandability and identical processor units, which make it more attractive for some applications.

We identified that the problem of communications overhead in shared I/O is serious enough to warrant the design of special controllers to manage the message traffic between adjacent processors. The HOLD controller and the FIFO circuitry, make use of the "WAIT" characteristic of Z80 microprocessor to accomplish the task. By monitoring certain control lines, they are able to temporarily halt the processors on exceptional conditions thus relieving the processors from testing for such situations themselves. The design is reasonably straight forward and the final outcome is an assembly of simple circuit modules such as NAND gates and flipflops except for the FIFO circuit which uses a LSI Am2812 first-in, first-out buffer. With little modification, they can function as separate peripheral attachments to the processors to and from which to send and receive data. We note in passing that the HOLD controller and the FIFO circuit could also be used to interface two stages in a pipeline if desired.

5.2. Future work

(1) The design of the simple interconnection scheme assumes that the communication needs are localized; that is, non-local values required by one loop repetition is produced by the previous repetition. If repetition I produces values to be used in repetition J, J > I+1, then a more generalized interconnection scheme may be required. This is necessary to route the data directly to the processors that need them, without the necessity of having to move them through a chain of intermediate processors. The reason is the obvious overheads incurred by all the processors in passing data that is not useful to them to others in the network. However, a generalized interconnection network is complex and hence expensive. Clearly a design tradeoff exists here. It is an interesting problem to investigate the advantages of using software to manage a part of the communication and hardware for the rest.

(2) More algorithms should be investigated to examine their suitability for implementation on shared I/O. Kung reports a number of algorithms for VLSI implementation [KUNG79]. Most of the algorithms under the

category of one dimensional linear array[1] algorithms such as discrete Fourier Transform could probably implemented on shared I/O. These algorithms are executed in a pipelined manner and if implemented in shared I/O, could benefit from some of the features of shared I/O discussed in this thesis.

[1] In a one-dimensional liner array, processors are arranged in a linear fashion just as in shared I/O.

References

AMDI80 Advanced Micro Devices, Inc, MOS/LSI Data Book

ANDE67 Anderson D.W., F.J.Sparcio, R.M.Tomasulo: "The IBM system 360, model 91: Machine philosophy & instruction handling", *IBM Journal of Research & Development*, Vol 2, January 1967, page 8-25.

BARN68 Barnes G.H, et al.: "The ILLIAC IV Computer" *IEEE Transactions on Computers*, Vol C-17, August 1968.

BASK72 Baskin H.B., B.R.Borgerson, R.Roberts: "PRIME - A modular architecture for terminal oriented systems", *Proceedings of AFIPS 1972 Spring Joint Computer Conference*, Vol 41, 1972, page 431-437.

BATC80 Batcher, K.E: "Design of massively parallel processor", *IEEE Transactions on Computers*, Vol C-29, September 1980, page 836-840.

BATC82 Batcher, K.E: "Bit-serial parallel processing systems", *IEEE Transactions on Computers*, Vol C-31, May 1982, page 377-384.

BERN82 Bernard, R: "Giants in small packages", *IEEE Spectrum*, Vol 19, No 2, February 1982.

BROO76 Brooks R.A., Giovanni Di Chiro: "Principles of Computer Assisted Tomography in Radiographic and Radioisotopic Imaging", *Phys.Med.Biol.*, Vol 21, No 5, 1976, page 689-732.

ENSL74 Enslow P.H Jr: "Multiprocessor Organization - A survey", *ACM Computing Surveys*, Vol 10, No 1, March 1977.

ENSL75 Enslow P.H Jr (ed): "Multiprocessors and Parallel Processing", John Wiley & Son, New York 1974.

ENSL78 Enslow P.H Jr: "What is a 'distributed' data processing system", *Computer*, Vol , 1978, page 13-21.

EVEN73 Evensen A.J., J.L.Troy: "Introduction to the Architecture of a 288 Element PEPE", *Proceedings of* 1973 *Sagamore Computer Conference on Parallel Processing*, 1973, page 162-169.

FENG81 Feng, Tse-yun: "A survey of interconnection networks", *Computer*, Vol 14, No 12, December 1981, page 12-27.

GABL80 Gable, M.G: "Communications in distributed systems. Part I - interfacing techniques", *Computer Design*, February 1980.

HAYN82 Haynes, L.S., R.L.Lau, D.P.Siewiorek, D.W.Mizell: "A survey of highly parallel computing", *Computer*, Vol 15, No 1, January 1982.

HEUF80 Heuft R.W: "Multiprocessor design for a class of dedicated applications", P.hD dissertation, Department of Electrical Engineering, University of Waterloo, 1980.

JENS78 Jensen E.D.: "The Honeywell Experimental Distributed Processor - An Overview", *Computer*, Vol 11, January 1978, page 28-37.

KOGG81 Kogge, P.M: "The architecture of pipelined computers", McGraw-Hill, 1981.

KUNG79 Kung H.T: "Let's design algorithms for VLSI systems", Technical report, CMU-CS-79-151, Department of Computer Science, Carnegie-Mellon University, 1979.

KUNG80 Kung H.T., C.E.Leiserson: "Algorithms for VLSI processor arrays", in *Introduction to VLSI systems*, by Mead C., L.Conway, Addison-Wesley publishing company, 1980.

KUNG82 Kung H.T: "Why systolic architectures?", *Computer*, Vol 15, No 1, January 1982.

LEE 80 Lee R.B: "Empirical results on the speed, efficiency, redundancy and quality of parallel computations" *Proceedings of the* 1980 *International conference on parallel processing*, IEEE, 1980.

MOST79 Microcomputer Data book, Mostek 1979.

ONOE81 Onoe M., K.Preston, A.Rosenfeld: "Real-Time/Parallel Computing", Plenum Press, New York, 1981.

OSBO78 Osborne A., G.Kane: "An introduction to microcomputers. Volume 2. Some real microprocessors", Adam Osborne & Associates, Inc., Berkely, 1978.

OSBO81 Osborne A., G.Kane: "Osborne 16-bit microprocessor handbook", Adam Osborne & Associates, Inc., Berkely, 1981.

RABI75 Rabiner L.R., B.Gold: "Theory and Applications of Digital Signal Processing", Prentice-Hall, Inc., Englewood Cliff, New Jersey, 1975.

RAMA77 Ramamoorthy C.V., H.F.Li: "Pipeline Architecture", *ACM Computing Surveys*, Vol 9, No 1, March 1977, page 61-102.

RUDO72 Rudolph J.A: "A production implementation of an associative array processor: STARAN", *Proceedings of AFIPS 1972 Fall Joint Computer Conference*, Vol 41, 1972, page 229-241.

SATY80 Satyanarayanan M: "Multi-processors, A comparative study", Prentice-Hall, Inc, New Jersey, 1980.

SROD78 Srodawa R.J: "Positive experiences with a multiprocessing system", *ACM Computing Surveys*, Vol 10, No 1, March 78, page 73-82.

SWAN77 Swan, R.J., S.H.Fuller, D.P.Siewiorek: "Cm* - A modular multi-microprocessor", *National Computer Conference*, 1977, page 637-644.

SWAR82 Swartzlander, E.E., B.K.Gilbert: "Supersystems: technology and architecture", *IEEE Transactions on Computers*, Vol C-31, May 1982, page 377-384.

THUR72 Thurber, K.J., E.D.Jensen, L.A.Jack, L.L.Kinney, P.C.Patton, L.C.Anderson: "A systematic approach to the design of digital bussing structures", *Proceedings of AFIPS 1972 Fall Joint Computer Conference*, Vol 41, 1972, page 719-740.

TREA82 Treleven P.C., D.R.Brownbridge, R.P.Hopkins: "Data-driven and Demand-driven Computer Architectures", *Computing Surveys*, Vol 14, No 1, March 1982.

UNGE58 Unger S.H: "A Computer Oriented Toward Spatial Problems", *Proceedings of the IRE*, October 1958, page 1744.

WATS72 Watson, W.J: "The TI-ASC - A highly modular and flexible supercomputer architecture", *Proceedings of AFIPS* 1972 *Fall Joint Computer Conference*, Vol 41, 1972, page 221-228.

WULF72 Wulf W.A., C.G.Bell: "C.mmp - A multi-mini-processor", *Proceedings of AFIPS* 1972 *Fall Joint Computer Conference*, Vol 41, 1972, page 765-777.